(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
10 April 2003 (10.04.2003)

PCT

(10) International Publication Number
WO 03/030031 A2

(51) International Patent Classification[7]: G06F 17/30

(21) International Application Number: PCT/US02/30783

(22) International Filing Date:
27 September 2002 (27.09.2002)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/326,052 28 September 2001 (28.09.2001) US
60/378,800 7 May 2002 (07.05.2002) US

(71) Applicant: **ORACLE INTERNATIONAL CORPORATION** [US/US]; 500 Oracle Parkway, M/S 5op7, Redwood Shores, CA 94065 (US).

(72) Inventors: **MURTHY, Ravi**; 33227 Jamie Circle, Fremont, CA 94555 (US). **KRISHNAPRASAD, Muralidhar**; 1065 Foster City Boulevard, Unit D, Foster City, CA 94404 (US). **CHANDRASEKAR, Sivasankaran**; 170 Waverly Street, Apt. F, Palo Alto, CA 94301 (US). **SEDLAR, Eric**; 4270 Cesar Chavez Street, San Francisco, CA 94131 (US). **KRISHNAMURTHY, Viswanathan**; 4735 Touchstone Terrace, Fremont, CA 94555 (US). **AGARWAL, Nipun**; 4768 Cheeney St., Santa Clara, CA 94131 (US).

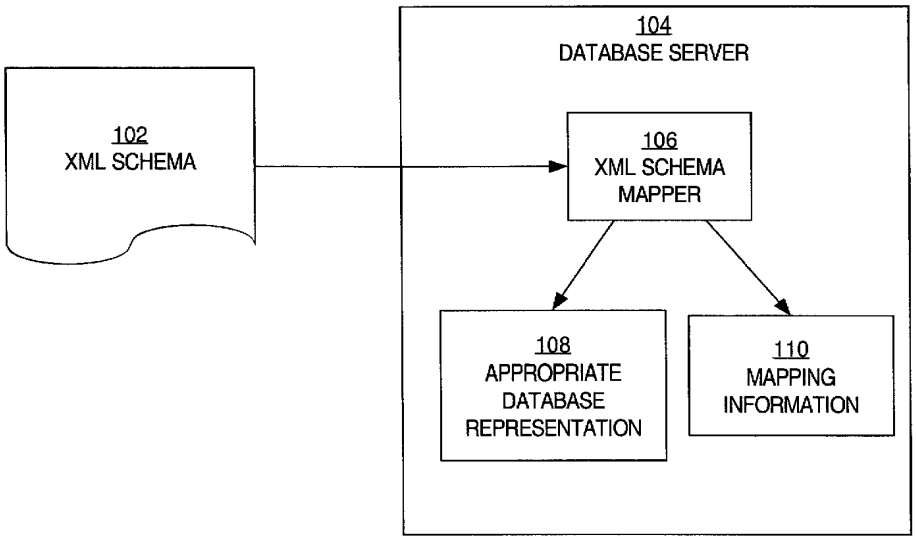
(74) Agents: **HICKMAN, Brian** et al.; 1600 Willow Street, San Jose, CA 95125 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, SK,

[Continued on next page]

(54) Title: MECHANISM FOR MAPPING XML SCHEMAS TO OBJECT-RELATIONAL DATABASE SYSTEMS



(57) Abstract: A method and system are provided for allowing users to register XML schemas in a database system. The database system determines, based on a registered XML schema, how to store within the database system XML documents that conform to the XML schema. This determination involves mapping constructs defined in the XML schema to constructs supported by the database system. Such constructs may include datatypes, hierarchical relationship between elements, constraints, inheritances, etc. Once the mapping has been determined, it is stored and used by the database system to determine how to store subsequently received XML documents that conform to the registered XML schema.

WO 03/030031 A2

TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

MECHANISM FOR MAPPING XML SCHEMAS TO OBJECT-RELATIONAL DATABASE SYSTEMS

RELATED APPLICATIONS

The present application claims priority from the following U.S. Provisional Patent Applications, the entire contents of which are incorporated herein by reference for all purposes:

U.S. Provisional Patent Application No. 60/326,052, filed on September 28, 2001, entitled File Based Access Provided With a Database System, by Eric Sedlar and Viswanathan Krishnamurthy;

U.S. Provisional Patent Application No. 60/378,800, filed on May 7, 2002, entitled SQL Access to Data that Provides a File System Abstraction, by Nipun Agarwal, Ravi Murthy, Eric Sedlar, Sivasankaran Chandrasekar, Fei Ge, Syam Pannala, Neema Jalali and Muralidhar Krishnaprasad.

The present application is also related to the following U.S. Patent Applications, the entire contents of which are incorporated herein by reference for all purposes:

U.S. Patent Application Serial No. _____, filed on the equal day herewith, entitled OPERATORS FOR ACCESSING HIERARCHICAL DATA IN A RELATIONAL SYSTEM, by Nipun Agarwal, Ravi Murthy, Eric Sedlar, Sivasankaran Chandrasekar and Fei Ge (Attorney Docket No. 50277-1975);

U.S. Patent Application Serial No. _____, filed on the equal day herewith, entitled PROVIDING A CONSISTENT HIERARCHICAL ABSTRACTION OF RELATIONAL DATA, by Nipun Agarwal, Eric Sedlar, Ravi Murthy and Namit Jain (Attorney Docket No. 50277-1976);

U.S. Patent Application Serial No. _____, filed on the equal day herewith, entitled INDEXING TO EFFICIENTLY MANAGE VERSIONED DATA IN A DATABASE SYSTEM , by Nipun Agarwal, Eric Sedlar and Ravi Murthy (Attorney Docket No. 50277-1978);

U.S. Patent Application Serial No. _____, filed on the equal day herewith, entitled MECHANISMS FOR STORING CONTENT AND PROPERTIES OF HIERARCHICALLY ORGANIZED RESOURCES, by Ravi Murthy, Eric Sedlar, Nipun Agarwal, and Neema Jalali (Attorney Docket No. 50277-1979);

U.S. Patent Application Serial No. _____, filed on the equal day herewith, entitled MECHANISM FOR UNIFORM ACCESS CONTROL IN A DATABASE SYSTEM, by Ravi Murthy, Eric Sedlar, Nipun Agarwal, Sam Idicula, and Nicolas Montoya (Attorney Docket No. 50277-1980);

U.S. Patent Application Serial No. _____, filed on the equal day herewith, entitled LOADABLE UNITS FOR LAZY MANIFESTATION OF XML DOCUMENTS by Syam Pannala, Eric Sedlar, Bhushan Khaladkar, Ravi Murthy, Sivasankaran Chandrasekar, and Nipun Agarwal (Attorney Docket No. 50277-1981);

U.S. Patent Application Serial No. _____, filed on the equal day herewith, entitled MECHANISM TO EFFICIENTLY INDEX STRUCTURED DATA THAT PROVIDES HIERARCHICAL ACCESS IN A RELATIONAL DATABASE SYSTEM, by Neema Jalali, Eric Sedlar, Nipun Agarwal, and Ravi Murthy (Attorney Docket No. 50277-1982).

FIELD OF THE INVENTION

The present invention relates to techniques for storing XML data in a database system.

BACKGROUND OF THE INVENTION

Within a relational database system, data is stored in various types of data containers. Such data containers typically have a structure. The structure of a container is imposed on the data it contains. For example, tables are organized into rows and columns. When data is stored in a table, individual data items within the data are stored in the specific rows and columns, thus imposing a structure on the data.

Typically, the structure imposed on the data corresponds to logical relationships within the data. For example, all values stored within a given row of a table will typically have some logical relationship to each other. For example, all values within a given row of an employee table may correspond to the same employee.

Outside of database systems, the degree to which electronic data is structured may vary widely based on the nature of the data. For example, data stored in spreadsheets is generally highly structured, while data representing visual images is generally highly unstructured.

XML (eXtensible Markup Language) is becoming increasingly popular as the format for describing and storing all forms of data. Thus, providing support for storing,

searching and manipulating XML documents is an extremely important problem for data management systems today.

Information about the structure of specific types of XML documents may be specified in documents referred to as "XML schemas". For example, the XML schema for a particular type of XML document may specify the names for the data items contained in that particular type of XML document, the hierarchical relationship between the data items contained in that type of XML document, datatypes of the data items contained in that particular type of XML document, etc.

Unfortunately, although XML documents are structured, the structure of XML documents is largely ignored by database systems when database systems are used to store XML documents. For example, a highly structured XML document, containing multiple values for multiple attributes, may simply be stored as if it were an atomic undifferentiated piece of data in a single CLOB column of a table. When XML documents are stored in this fashion, the performance and scalability features of the database cannot be fully exploited to access the XML data.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

FIG. 1 is a block diagram of a database system that includes a mechanism for mapping constructs contained in XML schemas to object-relational constructs, according to an embodiment of the invention;

FIG. 2 is a block diagram illustrating a computer system on which embodiments of the present invention may be implemented;

FIG. 3 is a block diagram showing syntax for creating an XML type table, according to an embodiment of the invention;

FIG. 4 is a block diagram showing a database system configured to create database objects for an appropriate database representation for documents conform to a particular XML schema, according to an embodiment of the invention;

FIG. 5 is a block diagram showing that XML strings are selectively mapped to two alternative database-supported datatypes;

FIG. 6 shows a complexType being mapped to SQL for out-of-line storage;

FIG. 7 shows complexType XML fragments mapped to character large objects (CLOBs);

FIG. 8 showis cross-referencing between complexTypes in the same XML schema;

FIG. 9 is a block diagram showing complexType self-referencing within an XML schema; and

FIG. 10 is a block diagram showing cyclical references between XML schema.

DETAILED DESCRIPTION OF EMBODIMENTS OF THE INVENTION

A method and system are described for mapping XML schemas to object-relational database systems. In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

FUNCTIONAL OVERVIEW

Various techniques are described herein for managing XML data within a database system in a manner that increases the correlation between the structure imposed on the data by the database containers used to hold the data, and the structure of the XML documents from which the data originates. According to one aspect, a mechanism is provided to allow users of a database system to register XML schemas with the database system. An XML schema may be registered explicitly (via an API call) or implicitly (when an instance document conforming to the XML schema is first inserted into the database).

During the registration process for a given XML schema, the database system determines (1) an appropriate database representation for the XML schema and (2) mapping information. The "appropriate database representation" determination is a determination about how data that conforms to the XML schema should be managed by the database system. Determining the appropriate database representation for a given XML schema may involve, for example, determining the database objects, collection types, constraints, and even the indexes that are to be used by the database system to store data from XML documents that conform to the given XML schema.

The mapping information indicates the mapping between the constructs included in the XML schema and the constructs included in the appropriate database representation. The mapping information may indicate, for example, that data associated with a specific element of the XML schema should be stored in a particular column of a table that is generated as part of the appropriate database representation. Typically, the appropriate database representation and the mapping information are generated so as to create a high correlation between the structure described in the XML schema and the structure imposed on the data by the database containers in which the XML data is stored.

SYSTEM OVERVIEW

FIG. 1 is a block diagram of a system that includes a mechanism for mapping XML schemas to object-relational database systems. Specifically, a database server 104 (also referred to herein as "XDB") includes an XML schema mapper 106. When an XML schema 102 is registered with database server 104, XML schema mapper 106 determines the appropriate database representation 108 for documents that conform to the XML schema 102, and generates mapping information 110 that indicates the correlation between the elements of the XML schema and the elements of the appropriate database representation 108.

According to one embodiment, database server 104 is configured to:

- Register any W3C compliant XML schema
- Perform validation of XML documents against a registered XML schema
- Register both local and global schemas
- Generate XML schemas from object types
- Support re-registering a XML schema (as a mechanism for manual schema evolution)
- Support implicit registration of XML schema when documents are inserted via certain APIs (e.g. FTP, HTTP)
- Allow a user to reference a schema owned by another user
- Allow a user to explicitly reference a global schema when a local schema exists with the same name.

- Support XML schema evolution

According to one embodiment, XML schema mapper 106 is configured to:

- Generate structured database mapping from XML Schemas (typically during schema registration) - this may include, for example, generation of SQL object types, collection types, etc and capturing the mapping information via schema annotations.
- Allow a user to specify a particular SQL type mapping when there are multiple legal mappings
- Create XMLType tables and columns based on registered XML schemas
- DML and query support for schema-based XMLType tables

XML SCHEMA REGISTRATION

According to one embodiment, an XML schema has to be first registered with database server 104 before it can be used or referenced within database server 104. After the registration process is completed, XML documents conforming to this schema (and referencing it via the schema URL within the document) can be handled by database server 104. Tables and/or columns can be created for root XML elements defined by this schema to store the conforming documents.

According to one embodiment, a schema is registered using a DBMS_XMLSCHEMA package by specifying the schema document and its URL (also known as schema location). Note that the URL used here is simply a name that uniquely identifies the registered schema within the database - and need not be the physical URL at which the schema document is located. Further, the target namespace of the schema is another URL (different from the schema location URL) that specifies an "abstract" namespace within which the elements and types get declared. An instance document should specify both the namespace of the root element and the location (URL) of the schema that defines this element.

For example consider the XML Schema shown below. It declares a complexType called "PurchaseOrderType" and an element "PurchaseOrder" of this type.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/PO.xsd">
```

```
  <complexType name = "PurchaseOrderType">
   <attribute name = "PurchaseDate" type = "date"/>
   <sequence>
    <element name = "PONum" type = "decimal"/>
    <element name = "Company" type = "string" maxLength = "100"/>
    <element name = "Item" maxOccurs = "1000">
      <complexType>
        <sequence>
          <element name = "Part" type = "string" maxLength = "1000"/>
          <element name = "Price" type = "float"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
 </complexType>

 <element name="PurchaseOrder" type="PurchaseOrderType"/>
</schema>
```

The following statement registers this schema at URL "http://www.oracle.com/PO.xsd". (doc is a variable holding the above schema text).

```
dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd'
, doc);
```

As shall be described in greater detail hereafter, a registered XML Schema can be used to create schema-based XMLType tables and columns. The following is an XMLType instance that conforms to the above XML schema. The schemaLocation attribute specifies the schema URL.

```
<PurchaseOrder xmlns="http://www.oracle.com/PO.xsd"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
               xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd"
               PurchaseDate="01-JAN-2001">
     <PONum>1001</PONum>
     <Company>Oracle Corp</Company>
     <Item>
       <Part>9i Doc Set</Part>
```

```
        <Price>2550</Price>
      </Item>

</PurchaseOrder>
```

According to one embodiment of the invention, XML schema registration includes (1) schema validation, (2) determination of appropriate data structures, and (3) generation of mapping information. Each of these phases shall be described in greater detail hereafter.

XML SCHEMA VALIDATION

XML schemas describe the structure of a particular type of XML document. However, XML schemas are themselves XML documents that must conform to the structure specified in an XML schema. Specifically, each XML schema must conform to the structure described in the XML schema document associated with the XML schema document type. During the schema validation phase of XML schema registration, the XML schema that is being registered is inspected to verify that the XML schema conforms to the structure specified in the XML schema associated with the XML schema document type.

DETERMINATION OF APPROPRIATE DATABASE REPRESENTATION

As mentioned above, the appropriate database representation determination is a determination about how data that conforms to an XML schema should be managed by the database system. According to one embodiment, the appropriate database representation is selected to achieve a high correlation between (1) the structure imposed on data by the XML document in which the data is contained, and the (2) the structure imposed on the data by a database system.

The ability to achieve a high correlation depends, at least in part, on the capabilities of the database system. The specific capabilities of database systems vary from vendor to vendor and version to version. While certain capabilities are common to most database systems, other capabilities are not. Thus, while embodiments of the present invention shall be described herein in the context of a database system with a specific set of capabilities, the invention is not limited to database systems that possess those specific capabilities.

According to one embodiment, the determination of the appropriate database representation is performed based on a set of general rules, governing the operation of XML schema mapper 106, about how to map each type of construct that may be encountered in an XML schema to a corresponding construct supported by the target object-relational database system. The rules may be hard-coded into the logic of XML schema mapper 106, or represented in metadata that is used by XML schema mapper 106. According to one embodiment, the general rules address the following issues:

- How to map datatypes supported by XML to datatypes supported by the target object-relational database system;
- How to map the structure defined by an XML schema to a database object with a similar structure;
- How to map constraints supported by XML to constraint enforcing mechanisms supported by the target object-relational database system;
- How to reflect, in the target object-relational database system, that the XML schema inherits from another XML schema; and
- How to reflect, in the target object-relational database system, other constructs supported by XML, such as substitution groups, simple content, wildcards, external references via include and import elements, etc.

MAPPING XML DATATYPES TO OBJECT-RELATIONAL DATATYPES

An XML schema declares a set of primitive types. According to one embodiment, the rules used by the XML schema mapper 106 define the datatypes, supported by the target database system, to which each of the XML datatypes correspond. For example, in one embodiment, the XML datatype "string" maps to either of VARCHAR or CLOB SQL datatypes. In this example, the XML schema mapper 106 may choose whether to map a particular string element to a VCHAR or CLOB based, for example, on any length constraints that could be declared, for the string element, in the XML schema. Numerous examples of the datatype-to-datatype mapping rules that XML schema mapper 106 may use are presented hereafter, and described in Appendix I.

MAPPING XML STRUCTURE TO DATABASE OBJECTS

SQL schemas describe the structure of an element in terms of the elements and attributes that can appear within it. The rules that map XML structure to database objects indicate how to map an SQL object type with attributes corresponding to the

XML attributes and elements defined within the XML schema. For example, an XML element A containing attribute X and elements Y and Z, will map to an object type with three attributes: X, Y and Z.

MAPPING XML CONSTRAINTS TO DATABASE CONSTRAINTS

XML schemas can specify various forms of constraints. Such constraints, when encountered by XML schema mapper 106, are mapped to the appropriate constraint mechanisms in SQL. For example, the length constraint for a "string" attribute defined in an XML schema may be maxLength="20". According to one embodiment, such a constraint would cause the string attribute to be mapped to the data type VARCHAR2(20).

Another type of constraint that can be applied to XML elements is a constraint that specifies a maximum number of occurrences of the element. When the maximum number is greater than one, the element can be mapped to an array type supported by the target database system (e.g. VARRAY). The number of occurrences specified for the XML constraint dictates the cardinality of the VARRAY.

Other types of constraints that may be specified for elements of an XML schema, and reflected in corresponding constraints in the appropriate database representation, include uniqueness constraints, referential integrity constraints, not null constraints, etc.

MAPPING INHERITANCE

The XML schema model allows for inheritance of complex types. According to one embodiment, when an XML schema makes use of the inheritance construct, the inheritance is mapped to the SQL object inheritance mechanisms supported by the target database system. For example, within an XML schema, an XML complexType "USAddress" can be declared as an extention of another complexType "Address". In response, within the appropriate database representation, an SQL object type "USAddress" is declared as a subtype of the SQL object type that corresponds to "Address".

LOCAL AND GLOBAL SCHEMAS

By default, an XML schema belongs to the user performing the registration. A reference to the XML schema document is stored within the XDB hierarchy within the directory /sys/schemas/<username>/.... For example, if the user SCOTT registered the

above schema, it gets mapped to the file
/sys/schemas/SCOTT/www.oracle.com/PO.xsd
Such schemas are referred to as local schemas. In general, they are usable only by the user to whom it belongs. Note that there is no notion of qualifying the schema URL with a database user name, because the schema location appearing in instance XML documents are simply URLs. Thus, only the owner of the schema can use it in defining XMLType tables, columns or views, validating documents, etc.

In contrast to local schemas, privileged users can register a XML schema as a global schema - by specifying an argument to dbms_xmlschema registration function. Global schemas are visible to all users and are stored under /sys/schemas/PUBLIC/.... directory within the XDB hierarchy. Note that the access to this directory is controlled by ACLs - and by default, is write-able only by DBA. A user needs to have write privileges on this directory to be able to register global schemas.

A user can register a local schema with the same URL as an existing global schema. A local schema always hides any global schema with the same name(URL).

A user can register a link to an existing schema - potentially owned by some other user. The schema link is identified by its URL. The schema link URL can then be used wherever a schema URL is expected. e.g. creating a xmltype table. The reference to the schema link gets translated to the underlying schema at the time of reference.
If a user has a local schema with the same name as a global schema, there is a mechanism that allows the user to explicitly reference the global schema. The user can register a link (with a different name) to the global schema.

DELETING XML SCHEMAS

According to one embodiment, an XML Schema can be deleted by using the dbms_xmlschema.deleteSchema procedure. When a user tries to delete a schema, the database server first checks for its dependents. If there are any dependents, the database server raises an error and the deletion operation fails. A FORCE option is provided while deleting schemas - if the user specifies the FORCE option, the schema deletion will proceed even though it fails the dependency check. In this mode, schema deletion will mark all its dependents as invalid.

DEPENDENCY MODEL FOR XML SCHEMAS

According to one embodiment, the following objects "depend" on a registered XML schema:

- Tables/Views that have a XMLType column that conforms to some element in this schema.
- XML schemas that include or import this schema as part of their definition
- Cursors that reference the schema name for eg. within XMLGEN operators. (Note:These are purely transient objects)

The following operations result in dependencies being added on a XML schema object :

- Schema registration : Add dependencies on all included/imported schemas Table/View/Cursor creation : Add dependency from table/view/cursor on the referenced xml schema object.

TRANSACTIONAL BEHAVIOR

According to one embodiment, the registration of a schema is non-transactional and auto-committed similar to other SQL DDL operations. If the registration is successful, the operation is auto-committed. However, if the registration fails, the database is rolled back to the state before the registration began. Since the schema registration process potentially involve creating object types and tables, the error recovery involves dropping any such created tables and types. Thus, the entire schema registration is guaranteed to be atomic i.e. it either succeeds or else the database is restored to the state before the start of registration.

XML SCHEMA EVOLUTION

A user may evolve a registered XML schema by re-registering it and providing the new XML schema document. The dbms_xmlschema.registerSchema function can be used to re-register the XML schema. This operation always succeeds if there are no XMLType tables that depend on this schema (XMLType views are okay). According to one embodiment, if there are any dependent XMLType tables, database server 104

requires that the input schema document contain the complete SQL mapping annotations - and that they represent a valid mapping applicable to all such XMLType tables.

Example - Changing the names of elements or attributes: The user retrieves the registered schema document, makes the needed modifications and re-registers it. Note that this alteration does not affect the underlying tables.

Example - Adding a new element or attribute: Since this alteration affects underlying tables, it has to be performed in multiple steps. The user first uses the ALTER TYPE and/or ALTER TABLE commands to evolve the underlying tables. This marks the XML schema as invalid. The user then modifies the XML schema document as appropriate and re-registers it.

According to one embodiment, a 1-step XML schema evolution is provided, i.e. a user simply inputs a new XML schema and all underlying type and table alterations are determined implicitly.

IMPLICIT REGISTRATION OF XML SCHEMAS

When instance documents are inserted into XDB via protocols such as HTTP or FTP, the schemas to which they conform (if specified) are registered implicitly - if not already registered. Since the schema registration is always auto-committed, the implicit registration is performed within an autonomous transaction.

XMLTYPE TABLES

Tables and columns that are part of the "appropriate database representation" of an XML schema are referred to herein as "schema-based" tables and columns. According to one embodiment, Schema-based XMLType tables and columns can be created by referencing the schema URL (of a registered schema) and the name of the root element. A subset of the XPointer notation (shown below) can also be used in providing a single URL containing both the schema location and the element name.

```
CREATE TABLE po_tab OF xmltype
 XMLSCHEMA "http://www.oracle.com/PO.xsd" ELEMENT
"PurchaseOrder"

An equivalent definition is

    CREATE TABLE po_tab of xmltype
       element "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

By default, schema-based XMLType is stored in an underlying (hidden) object type column. The SQL object types can be created (optionally) during the schema registration process. The mapping from XML to SQL object types and attributes is itself stored within the XML schema document as extra annotations i.e. new attributes defined by XDB.

Schema-based XMLType can also be stored in a single underlying LOB column.

```
CREATE TABLE po_tab OF xmltype
   STORE AS CLOB
   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

Creation of SQL object types

According to one embodiment, when an XML schema is registered, database server 104 creates the appropriate SQL object types that enable a structured storage of XML documents conforming to this schema. All SQL object types are created in the current user's schema (by default). For example, when PO.xsd is registered, the following SQL types are created.

```
create type Item_t as object
(
  part varchar2(1000),
  price number
);
create type Item_varray_t as varray(1000) of OBJ_T1;
create type PurchaseOrder_t as object
(
  purchasedate date,
  ponum number,
  company varchar2(100),
  item Item_varray_t
);
```

The names of the object types and attributes above may actually be system-generated. If the schema already contains the SQLName attribute filled in, this name is used as the object attribute's name. Else, the name is derived from the XML name - unless it cannot be used because of length, or conflict reasons. If the SQLSchema attribute is filled in, Oracle will attempt to create the type in the specified schema. The current user must have any necessary privileges to perform this operation.

MAPPING XML SCHEMAS TO OBJECT TYPES – A DETAILED EXAMPLE

The following sections provide the details on how the SQL object types may be generated from the XML schema information. As was mentioned above, the actual mapping rules may vary from implementation to implementation based on a variety of factors. One such factor is the capabilities of the target database system. In the following detailed example, it is assumed that the target database system supports the data types and object typing mechanisms currently available in the Oracle 9iR2, currently available from Oracle Corporation.

MAPPING SIMPLE TYPES

According to one embodiment, an XML primitive type is mapped to the closest SQL datatype. For example, decimal, positive Integer and float are all mapped to SQL NUMBER. An XML enumeration type is mapped to an object type with a single RAW(n) attribute - the value of n is determined by the number of possible values in the enumeration declaration. An XML list or union datatype is mapped to a string (VARCHAR2/CLOB) datatype in SQL.

Default mapping of XML simple types to SQL

XML simple type	Length/ Precision	MaxLength / Scale	Default Oracle datatype	Compatible datatypes	Comments
string		n (n < 4000)	VARCHAR2(n)	NVARCHAR2, CHAR, CLOB, NCHAR, NCLOB	n < 2000 for UTF-1 encoding
string	m (m < 4000)		CHAR(n)	VARCHAR2, NVARCHAR2, CLOB, NCHAR, NCLOB	n < 2000 for UTF-1 encoding
string	m (m > 4000)		CLOB	VARCHAR2, NVARCHAR2, CHAR,	n > 2000 for UTF-1 encoding

				NCHAR, NCLOB	
string		n (n > 4000)	CLOB	VARCHAR2, NVARCHAR2, CHAR, NCHAR, NCLOB	n > 2000 for UTF-1 encoding
boolean			RAW(1)		Values MUST be 0 1.
float			FLOAT	NUMBER, DOUBLE	
double			DOUBLE	NUMBER	
decimal	precision m	scale n	NUMBER(m,n)		If m & n == 0, then map to NUMBER
timeInstant			TIMESTAMP	TIME??	Timstamp in XML can map to the form CCYY-MM-DDTh mm-ss.sss
timeDuration			INTERVAL	TIMESTAMP??	XML duration supports PnYnMnDTnHnMr format from ISO 8601.
recurringDuration			INTERVAL		
binary	m (m < 4K)	OR n, n<4K	RAW(m) or RAW(n)	BLOB	
binary	m (m > 4K)	OR n, n>4K	BLOB	RAW	length or maxlengt 4K
uri			UriType (VARCHAR2)	NVARCHAR2, CLOB, NCHAR,	Length or uri must l less than 4K Or 2K for UTF-16..

				CHAR, NCLOB	

Default XML Datatype Mapping to SQL (for simple Types) (CONTD)

XML simpleType	Default Oracle Type	Compatible Types	Comments
Language (string)	VARCHAR2(4000)	NVARCHAR2, CLOB,CHAR, NCLOB, NCHAR	2000 for UTF-16 (for all)
NMTOKEN (string)	VARCHAR2(4000)	-- same --	""
NMTOKENS (string)	VARCHAR2(4000)	-- same --	""
Name (string)	VARCHAR2(4000)	-- same --	A generic XML Name
NCName (string)	VARCHAR2(4000)	-- same --	Represents an non-colon'ized name
ID	VARCHAR2(4000)	-- same --	Unique throughout the document
IDREF	VARCHAR2(4000)	-- same --	Must match an ID in the document
IDREFs	VARCHAR2(4000)	-- same --	
ENTITY	VARCHAR2(4000)	-- same --	
ENTITIES	VARCHAR2(4000)	-- same --	
NOTATION	VARCHAR2(4000)	-- same --	
QName	XDB.XDB$QNAME		Represents a qualified XML name. Stored an an object type with two attributes - the unqualified name string and the index number of

			the namespace into the global namespace array.
integer	INTEGER	INT, NUMBER	
nonNegativeInteger	INTEGER	INT, NUMBER	
positiveInteger	INTEGER	INT, NUMBER	
nonPositiveInteger	INTEGER	INT, NUMBER	
negativeInteger	INTEGER	INT, NUMBER	
date	DATE	TIMESTAMP	
time	TIMESTAMP	DATE	

MAPPING COMPLEX TYPES

According to one embodiment, a complextype is mapped to an object type. XML attributes declared within the complexType map to object attributes - the simpleType defining the XML attribute determines the SQL datatype of the corresponding attribute. XML elements declared within the complexType are also mapped to object attributes. The datatype of the object attribute is determined by the simpleType or complexType defining the XML element.

If the XML element is declared with maxOccurs attribute's value > 1, it is mapped to a collection attribute in SQL. The collection could be either a VARRAY (default) or nested table (if the maintainOrder attribute is set to FALSE). Further, the default storage of the VARRAY is in tables (OCTs) [OCT-FS] instead of LOBs - the user can choose the LOB storage by setting the storeAsLob attribute to TRUE.

In general, the name of the SQL attribute is generated from the XML element or attribute name using the following algorithm :

1. use XML element/attribute name (truncated to 30 chars)

2. if an illegal SQL character is found, map it to underscore ('_')

3. if this name is not unique, append a sequence number (note: this may require further truncating the name before appending the number)

However, the user can explicitly specify the SQL attribute name by providing a value for the SQLName attribute within the schema

DOM FIDELITY

All elements and attributes declared within the XML schema get mapped to separate attributes within the corresponding SQL object type. However, there are some pieces of information in the XML instance documents that are not represented directly by such element/attributes. Examples are :

- Comments
- Namespace declaration
- Prefix information

In order to guarantee that the returned XML documents are identical to the original document for purposes of DOM traversals (referred to as DOM fidelity), a binary attribute called SYS_XDBPD$ is added to all generated SQL object types. This attribute stores all pieces of information that cannot be stored in any of the other attributes - thereby ensuring DOM fidelity of XML documents stored in the database system. Note : The SYS_XDBPD$ attribute is omitted in many examples for reasons of clarity. However, the attribute is may be present in all SQL object types generated by the schema registration process.

SQL OUT OF LINE STORAGE

According to one embodiment, by default, a sub-element is mapped to an embedded object attribute. However, there may be scenarios where an out-of-line storage offers better performance. In such cases the SQLInline attribute can be set to FALSE - and the XML schema mapper 106 generates an object type with an embedded REF attribute. The REF points at another instance of XMLType that corresponds to the XML fragment that gets stored out-of-line. Default tables (of XMLType) are also created to store the out-of-line fragments.

Example

```
<complexType name = "Employee">  -- OBJ_T2
  <sequence>
    <element name = "Name" type = "string" maxLength = "1000"/>
    <element name = "Age" type = "decimal"/>
    <element name = "Addr" SQLInline = "false">
      <complexType>  -- OBJ_T1
```

```
        <sequence>
          <element name = "Street" type = "string" maxLength =
"100"/>
          <element name = "City" type = "string" maxLength =
"100"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
create type OBJ_T1 as object
(
  Street varchar2(100),
  City varchar2(100)
);
create type OBJ_T2 as object
(
  Name varchar2(100),
  Age number,
  Addr REF XMLType
);
```

MAPPING XML FRAGMENTS TO LOBS

A user can specify the SQLType for a complex element as LOB(CLOB/BLOB) in which case, the entire XML fragment gets stored in a LOB attribute. This is useful in scenarios where some portions of the XML document are seldom queried upon, but are mostly retrieved and stored as a single piece. By storing the fragment as a LOB, the parsing/decomposition/recomposition overhead is reduced.

Example

```
<complexType name = "Employee">  -- OBJ_T
  <sequence>
    <element name = "Name" type = "string" maxLength = "1000"/>
    <element name = "Age" type = "decimal"/>
    <element name = "Addr" SQLType = "CLOB">
      <complexType>
        <sequence>
          <element name = "Street" type = "string" maxLength =
"100"/>
```

```
            <element name = "City" type = "string" maxLength =
"100"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
  create type OBJ_T as object
  (
    Name varchar2(100),
    Age number,
    Addr CLOB
  );
```

MAPPING SIMPLE CONTENT

A complexType based on a simpleContent declaration is mapped to an object type with attributes that correspond to the XML attributes and an extra SYS_XDBBODY$ attribute corresponding to the body value. The datatype of the body attribute is based on the simpleType which defines the body's type.

Example

```
<complexType>
  <simpleContent>
    <restriction base = "string" maxLength = "1000">
      <attribute name = "a1" type = "string" maxLength = "100"/>
    </restriction>
  </simpleContent>
</complexType
create type OBJ_T as object
(
  a1 varchar2(100),
  SYS_XDBBODY$ varchar2(1000)
);
```

MAPPING ANY/ANYATTRIBUTE

any element declarations and *anyAttribute* attribute declarations are mapped to LOBs in the object type. The LOB stores the text of the XML fragment that matches the any declaration. The namespace attribute can be used to restrict the contents to belong to

a specified namespace. The *processContents* attribute within the *any* element declaration indicates the level of validation required for the contents matching the any declaration.

Example

```
<complexType name = "Employee">
 <sequence>
    <element name = "Name" type = "string" maxLength = "1000"/>
    <element name = "Age" type = "decimal"/>
    <any namespace = "http://www/w3.org/2001/xhtml"
processContents = "skip"/>
  </sequence>
</complexType>
create type OBJ_T as object
(
  Name varchar2(100),
  Age number,
  SYS_XDBANY$ blob
);
```

MAPPING STRINGS TO SQL VARCHAR2 VS CLOB

If the XML schema specifies the datatype to be "string" and a maxLength value of less than 4000, it gets mapped to a varchar2 attribute of the specified length. However, if the maxLength value is not specified in the XML schema, it can only be mapped to a LOB. This is sub-optimal in cases when the majority of string values are actually small - and a very small fraction of them is large enough to necessitate a LOB. The ideal SQL datatype would be varchar2(*) that would perform like varchars for small strings but can accommodate larger strings as well. Further, such columns should support all varchar functionality such as indexing, SQL functions, etc. A similar case can be made for needing a raw(*) datatype to hold unbounded binary values without loss of performance and/or functionality for the small cases.

According to an alternative embodiment, all unbounded strings are mapped to CLOBs and all unbounded binary elements/attributes are mapped to BLOBs.

MAPPING STRINGS TO SQL VARCHAR2 VS NVARCHAR2

By default, the XML string datatype is mapped to SQL varchar2. However, the user can override this behavior in a couple of ways :

1. The user can specify SQLType to be NVARCHAR2 for a particular string element or attribute. This ensures that NVARCHAR2 is chosen as the SQL type for the particular element/attribute.

2. The user can set the mapStringToNCHAR attribute to "true" at the top of the schema declaration. This ensures that all XML strings get mapped to NVARCHAR2 (or NCLOB) datatype, unless explicitly overridden at the element level.

CREATING SCHEMA-BASED XML TABLES

Assuming that the XML schema identified by "http://www.oracle.com/PO.xsd" has already been registered. A XMLType table can be created to store instances conforming to the PurchaseOrder element of this schema - in an object-relational format - as follows :

```
create table MyPOs of xmltype
   element "http://www.oracle.com/PO.xsd#PurchaseOder";
```

Hidden columns are created corresponding to the object type to which the PurchaseOrder element has been mapped. In addition, a XMLExtra object column is created to store the top-level instance data such as namespaces declarations, etc. ***Note : XMLDATA*** is a pseudo-attribute of XMLType that allows directly accessing the underlying object column.

SPECIFYING STORAGE CLAUSES

The underlying columns can be referenced in the storage clauses by

1. object notation : XMLDATA.<attr1>.<attr2>....

2. XML notation : ExtractValue(xmltypecol, '/attr1/attr2')

```
create table MyPOs of xmltype
element "http://www.oracle.com/PO.xsd#PurchaseOrder"
lob (xmldata.lobattr) store as (tablespace ...);
create table MyPOs of xmltype
element "http://www.oracle.com/PO.xsd#PurchaseOrder"
   lob (ExtractValue(MyPOs, '/lobattr')) store as (tablespace ...);
```

CREATING INDEXES

As shown above, columns underlying a XMLType column can be referenced using either a object notation or a XML notation in the CREATE INDEX statements.

```
create index ponum_idx on MyPOs (xmldata.ponum);
create index ponum_idx on MyPOs p (ExtractValue(p,
'/ponum');
```

CONSTRAINTS

Constraints can be specified for underlying columns by using either the object or the XML notation.

```
create table MyPOs of xmltype
element "http://www.oracle.com/PO.xsd#PurchaseOrder"
(unique(xmldata.ponum));
create table MyPOs p of xmltype
element
"http://www.oracle.com/PO.xsd#PurchaseOrder"(unique(ExtractV
alue(p , '/ponum'));
```

DMLS

New instances can be inserted into a XMLType table as :

```
insert into MyPOs values
(xmltype.createxml('<PurchaseOrder>.....</PurchaseOrder>'));
```

The XMLType table can be queried using the XPath-based SQL operators.

```
select value(p) from MyPOs where extractValue(value(p),
'/Company)     = 'Oracle';
```

The query rewrite mechanism rewrites queries involving existsNode and extract operators to directly access the underlying attribute columns - thereby avoiding construction of the XML followed by subsequent XPath evaluation. For example, the above query gets rewritten to :

```
select value(p) from MyPOs where p.xmldata.company =
'Oracle';
```

QUERY REWRITE

XPath based operators (Extract, ExistNode,ExtractValue) operating on schema-based XMLType columns are rewritten to go against the underlying SQL columns. This enables further SQL optimizations that fully exploit the object-relational storage of the XML. The following kinds of XPath expressions can be translated into the underlying SQL queries :

1. Simple XPath expressions - involving traversals over object type attributes only, where the attributes are simple scalars or object types themselves. The only axes supported are the child and the attribute axes.

2. Collection traversal expressions - involve traversal of collection expressions. Only axes supported are child and attribute axes.

3. Expressions involving * axes - Transform those expressions involving the wildcard axes provided the datatypes of the resulting nodes are all coercible. (e.g. CUST/*/CUSTNAME must point to CUSTNAMEs which are all of the same or coercible datatypes).

4. Expressions involving descendant axis (//) - Transform these expressions provided the datatypes of the resulting nodes are the same or coercible.

5. All of these expressions must work with the type cache, which includes "hidden" traversals like REFs to XMLTypes etc.. (for instance xdb$schema_t stores a varray of REFs to xdb$element_t and this is not directly apparent in the XPath expression or the resulting XML document).

Transformations of these XPath expressions are supported in the ExistsNode, ExtractValue and Extract usage scenarios.

Examples of query rewrite of XPath.

Original Query

```
select * from MyPOs p
   where ExistsNode(p, ?/PO[PNAME=?PO1?]PONO?) = 1
```

After Rewrite of ExistsNode

```
select * from MyPOs p
where (CASE WHEN (p.xmldata.pono IS NOT NULL)
            AND (p.xmldata.PNAME = ?PO1?)) THEN 1 ELSE 0 ) =
1
```

Original Statement

```
select ExtractValue(p, ?/[PNAME=?PO1']/PONO?) from MyPOs p
```

After Rewrite of Extract

```
select (select p.xmldata.pono from dual where
p.xmldata.pname = ?PO1?)
from MyPOs ;
```

FUNCTION REWRITE RULES

EXTRACT, EXTRACTVALUE and EXISTSNODE can appear in the following positions

- In the select list, where clause predicate, group by and order by expressions in a SQL query.
- In the Index clause of a CREATE INDEX statement.

```
create index foo_index on foo_tab (extractvalue(xml_col,
'/PO/PONO'));
```

In all these cases, the EXISTSNODE and EXTRACT operator get replaced by their defininining underlying expressions. The XPath expressions must satisfy the conditions listed in the previous section for them to be rewritten.

In the index case, if replacing the whole operator tree results in a single column, then the index is turned into a BTree or a domain index on the column, rather than being a functional index.

REWRITE FOR OBJECT/SCALAR ATTRIBUTE TRAVERSALS

Simple XPath traversals are rewritten into object type accessors. Predicates are handled by putting them in the where clause. Any XPath child access over an object type is translated to an object attribute access on the underlying object type. For example A/B maps to a.b where A maps to the object type a and the XPath node B maps to the attribute of "a" named "b".

This rewrite is consistent at any level of the XPath expression, i.e. whether the XPath traversal occurs within a predicate, or a location path variable.

For example,

PO/CUSTOMER/CUSTOMERNAME becomes *"po"."cust"."custname"* (assuming PO maps to "po" etc..)

Predicates are handled by rewriting the predicate expression in the underlying object expressions.

In the simple case, for EXISTSNODE, the main location path traversal becomes a IS NOT NULL predicate, whereas for the EXTRACT case, this becomes the actual node being extracted.

EXISTSNODE(po_col, 'PO/CUSTOMER/CUSTOMERNAME')
becomes

CASE (WHEN ("po"."cust"."custname" IS NOT NULL) then 1 else 0)

Predicates are handled in a similar manner.For example, in the operator given below,

EXISTSNODE(po_col, 'PO/CUSTOMER[CUSTOMERNO=20]/CUSTOMERNAME')

the predicate, D = 20 is treated as if the user specified, (A/B/D = 20)

Thus the whole expression becomes,

CASE (WHEN ("PO"."CUST"."CUSTNAME" IS NOT NULL
AND ("PO"."CUST"."CUSTNO" = 20)) THEN 1 ELSE 0)

COLLECTION TRAVERSALS

The XPath expressions may also span collection constructs and the queries are still rewritten by using subqueries on the collection tables. For example,

EXISTSNODE(po_col, '/PO/lineitems[lineitemno=20]') is checking for the existance of lineitems in a purchase order where the lineitem number is 20. This becomes,

case(when (exists(select * from TABLE("po"."lineitems") where lineitemno = 20)) then 1 else 0)

DEFAULT TABLES

As part of schema registration, default tables can also be created. The default table is most useful in cases when XML instance documents conforming to this schema are inserted through APIs that do not have any table specification e.g. FTP, HTTP. In such case, the XML instance is inserted into the default table.

If the user has given a value for defaultTable attribute, the XMLType table is created with that name. Else, it gets created with some internally generated name. Further, any text specified as the tableStorage attribute is appended to the generated CREATE TABLE statement.

SPECIFYING THE INTERNAL MEMORY DATATYPE

The XML data is stored in a C structure within RDBMS memory. In general, the in-memory representation of the XML data is such that it tries to avoid datatype conversions at load time, and converts data only when accessed, since many parts of the document may not be accessed at all. As part of schema registration, the in-memory datatype is chosen based on the XML datatype - and this information is stored within the schema document using the memDatatype attribute. However, there are some scenarios in which an application may wish to override the default memory type in favor of a different in-memory representation.

Eg. the default memory representation of strings is "char" which keeps the string data in the database session character set. However, if this data is only consumed by a Java application that requires it in Fixed Width UCS-2 Unicode, it may be more performant to set the memDatatype to "JavaString". This ensures that database server 104 keeps the data directly in Java memory in Unicode format - thereby avoiding any format conversions or copies.

XML Datatype	Allowed Memory Datatypes	Description	Default
String	Char	Varying width character data in character set currently active for this session.	Yes
	JavaString	Fixed width UCS-2 Unicode allocated from JServer memory.	No
Integer	integer	Signed 8 byte native integer by default; if XML schema specifies max & min values, a smaller or unsigned datatype may be used	Yes

	number	Oracle number format	No
float	Float	Native maximum precision floating point; smaller value may be used if max & min are specified within range of smaller type	Yes
	number	Oracle number format	No

GENERATION OF MAPPING INFORMATION

Once the appropriate database representation has been determined for a particular XML schema, mapping information is generated to indicate the correlation between the elements of the appropriate database representation and the elements identified in the particular XML schema. For example, if the appropriate database representation for an XML schema for type "person" includes a table PERSON for storing the data items contained in person XML documents, then the mapping information would indicate a correlation between person XML documents and table PERSON.

In addition to the general correlation between an XML schema and a database schema object (such as a table), the mapping information may reflect correlations at much finer levels of granularity. For example, the mapping information may indicate which specific column of the PERSON table should be used to store each specific data item within person XML documents.

According to one embodiment, the information regarding the SQL mapping is itself stored within the XML schema document. During the registration process, the XML schema mapper 106 generates the SQL types (as shown above). In addition it adds annotations to the XML schema document to store the mapping information. Annotations are in form of new attributes. Example : The schema below shows the SQL mapping information captured via SQLType and SQLName attributes.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/PO.xsd">
  <complexType name = "PurchaseOrder">
   <attribute name = "PurchaseDate" type = "date" SQLName="PURCHASEDATE"
SQLType="DATE"/>
   <sequence>
    <element name = "PONum" type = "decimal" SQLName="PONUM"
SQLType="NUMBER"/>
```

```
    <element name = "Company" type = "string" maxLength = "100"
SQLName="COMPANY" SQLType="VARCHAR2"/>
    <element name = "Item" maxOccurs = "1000" SQLName="ITEM"
SQLType="ITEM_T" SQLCollType="ITEM_VARRAY_T">
      <complexType>
        <sequence>
          <element name = "Part" type = "string" maxLength = "1000"
SQLName="PART" SQLType="VARCHAR2"/>
          <element name = "Price" type = "float" SQLName="PRICE"
SQLType="NUMBER"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
 </complexType>
 <element name="PO" type="PurchaseOrder" SQLType="PURCHASEORDER_T"/>
</schema>
```

USER-SPECIFIED NAMES IN INPUT SCHEMA DOCUMENT

The user can specify the names of the SQL object types and its attributes by filling in the SQLName and SQLType attributes prior to registering the schema. If the SQLName and SQLType values are specified by the user, then the XML schema mapper 106 creates the SQL object types using these names. If these attributes are not specified by the user, an internal name-generation algorithm is used to generate the names. See Appendix for details on the name generation algorithm.

The table below lists all the annotations used within the schema to capture the SQL mapping information. Note that the user need not specify values for any of these attributes. The XML schema mapper 106 will fill in the appropriate values during the schema registration process. However, it is recommended that user specify the names of at least the top level SQL types - in order to be able to reference them later. All annotations are in form of attributes that can be specified within attribute and element declarations. These attributes belong to the XDB namespace : http://xmlns.oracle.com/xdb/XDBSchema.xsd

Table 1 : XDB attributes specifiable within element and attribute declarations

Attribute	Values	Default	Description

SQLName	Any SQL identifier	element name	This attribute specifies the name of the attribute within the SQL object that maps to this XML element.
SQLType	Any SQL type name	Name generated from element name	This property specifies the name of the SQL Type corresponding to this XML element or attribute. This could refer to a scalar or object type depending on the XML schema declaration.
SQLCollType	Any SQL Collection type name	Name generated from element name	This specifies the name of the SQL collection type corresponding to this XML element that has maxOccurs > 1.
SQLSchema	Any SQL user name	User registering XML schema	Name of database user owning the type specified by SQLType
SQLCollSchema	Any SQL user name	User registering XML schema	Name of database user owning the type specified by SQLCollType.
maintainOrder	true | false	True	If "true", the collection is mapped to a VARRAY. Else, the collection is mapped to a NESTED TABLE.
storeVarrayAsLob	true | false	True	If "true", the VARRAY is stored in a LOB. If "false", the varray is stored as a table (OCT).
SQLInline	true | false	true	If "true" this element is stored inline as an embedded attribute (or a collection if maxOccurs > 1). If "false", a REF (or collection of REFs if maxOccurs > 1) is

			stored. This attribute will be forced to "false" in certain situations (like cyclic references) where SQL will not support inlining.
maintainDOM	true | false	true	If "true", instances of this element are stored such that they retain DOM fidelity on output. This implies that all comments, processing instructions, namespace declarations, etc are retained in addition to the ordering of elements. If "false", the output need not be guaranteed to have the same DOM behavior as the input.
tableStorage	any valid storage clause text	NULL	This attribute specifies the storage clause that is appended to the default table creation statement. It is meaningful mainly for elements that get mapped to tables viz. top-level element declarations and out-of-line element declarations.
defaultTable	Any table name	Based on element name.	This attribute specifies the name of the table into which XML instances of this schema should be stored. This is most useful in cases when the XML is being inserted from APIs where table name is not specified e.g. FTP, HTTP.
defaultACL	Any URL pointing to a ACL document	NULL	This attribute specifies the URL of the ACL that should be applied by default to all instances of this element [Folder-FS].
isFolder	true | false	false	If true, instances of this element can be used as a folder (or container) within

			XDB [Folder-FS].
mapStringToNCHAR	true | false	false	If "true", all XML strings get mapped to NVARCHAR2 (or NCLOB) datatype, unless explicitly overridden at the element level. If "false", all XML string elements/attributes are mapped to varchar2 columns.
memDatatype	in-memory datatype	internal	This attribute can be used to override the default in-memory mapping of (simple) elements and attributes. See below for the table of allowed memory datatypes for a given XML datatype.

HYBRID STORAGE MODELS

According to one embodiment, the XML schema mapper 106 is implemented to support hybrid storage models in which the structure of some elements defined within the XML schema is maintained in the appropriate database representation, and the structure of other elements is not. For example, the most-often queried/updated portions of an XML document type may be mapped to object type attributes, while the rest of the portions of the XML document are stored together in a CLOB. According to one embodiment, the specific portions for with structure is to be maintained or not to be maintained are designated by pre-annotating the XML schema with appropriate mapping directives.

TRANSACTIONAL NATURE OF XML SCHEMA REGISTRATION

According to one embodiment, the XML schema registration is performed using the transaction support of database server 104 in a manner that allows executing compensating action to undo partial effects when errors are encountered during the schema registration operation.

HANDLING CYCLIC DEFINITIONS IN XML SCHEMAS

It is possible for XML schemas to include cycles. According to one embodiment, XML schema mapper 106 is configured to detect such cycles and break them by using

REFs while mapping to SQL object types. A detailed description of how REFs may be used to break cycles is provided in Appendix I.

STORING XML DOCUMENTS BASED ON THE MAPPING INFORMATION

After an XML schema for a particular document type has been registered with database server 104, XML documents that conform with the schema can be intelligently managed by database server 104. According to one embodiment, when a protocol indicates that a resource must be stored in a database managed by database server 104, database server 104 checks the document's file name extension for .xml, .xsl, .xsd, and so on. If the document is XML, a pre-parse step is performed, where enough of the resource is read to determine the XML `schemaLocation` and `namespace` of the root element in the document. This location is used to look for a registered schema with that `schemaLocation` URL. If a registered schema is located with a definition for the root element of the current document, then the default table specified for that element is used to store that resource's contents.

According to one embodiment, when an XML document is stored in a database server that supports the XML schema registration techniques described herein, the database server is able to validate the documents to verify that they confirm to the corresponding XML schema. The validation may include validation of both the structure and the datatypes used by the XML document.

Various other benefits are achieved through the use of the techniques described herein. For example, the schema registration process allows the database server to enforce the integrity constraints and other forms of constraints on the XML documents and the tables used to store them. In addition, the database server is able to create indexes on and partition XML tables based on XML data.

Because the structure of the XML documents is reflected in how the data from the XML documents are stored within the database, the tag information typically used to reflect the structure does not need to be stored along with the data. The ability to avoid storing some or all of the XML tags can result in a significant decrease in storage overhead, since the XML tags often form a large portion of the size of XML documents.

Other performance benefits are also made possible. For example, query performance may be improved by rewriting XPath queries to directly access the underlying columns. In addition, update performance may be improved by rewriting updates to directly update the underlying columns. Consequently, updating a portion of

the XML data from a stored document would not always require the rewriting the entire XML data for the stored document.

HARDWARE OVERVIEW

Figure 2 is a block diagram that illustrates a computer system 200 upon which an embodiment of the invention may be implemented. Computer system 200 includes a bus 202 or other communication mechanism for communicating information, and a processor 204 coupled with bus 202 for processing information. Computer system 200 also includes a main memory 206, such as a random access memory (RAM) or other dynamic storage device, coupled to bus 202 for storing information and instructions to be executed by processor 204. Main memory 206 also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor 204. Computer system 200 further includes a read only memory (ROM) 208 or other static storage device coupled to bus 202 for storing static information and instructions for processor 204. A storage device 210, such as a magnetic disk or optical disk, is provided and coupled to bus 202 for storing information and instructions.

Computer system 200 may be coupled via bus 202 to a display 212, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device 214, including alphanumeric and other keys, is coupled to bus 202 for communicating information and command selections to processor 204. Another type of user input device is cursor control 216, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor 204 and for controlling cursor movement on display 212. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

The invention is related to the use of computer system 200 for implementing the techniques described herein. According to one embodiment of the invention, those techniques are performed by computer system 200 in response to processor 204 executing one or more sequences of one or more instructions contained in main memory 206. Such instructions may be read into main memory 206 from another computer-readable medium, such as storage device 210. Execution of the sequences of instructions contained in main memory 206 causes processor 204 to perform the process steps described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus,

embodiments of the invention are not limited to any specific combination of hardware circuitry and software.

The term "computer-readable medium" as used herein refers to any medium that participates in providing instructions to processor 204 for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device 210. Volatile media includes dynamic memory, such as main memory 206. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus 202. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infra-red data communications.

Common forms of computer-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.

Various forms of computer readable media may be involved in carrying one or more sequences of one or more instructions to processor 204 for execution. For example, the instructions may initially be carried on a magnetic disk of a remote computer. The remote computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system 200 can receive the data on the telephone line and use an infra-red transmitter to convert the data to an infra-red signal. An infra-red detector can receive the data carried in the infra-red signal and appropriate circuitry can place the data on bus 202. Bus 202 carries the data to main memory 206, from which processor 204 retrieves and executes the instructions. The instructions received by main memory 206 may optionally be stored on storage device 210 either before or after execution by processor 204.

Computer system 200 also includes a communication interface 218 coupled to bus 202. Communication interface 218 provides a two-way data communication coupling to a network link 220 that is connected to a local network 222. For example, communication interface 218 may be an integrated services digital network (ISDN) card or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface 218 may be a local area

network (LAN) card to provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface 218 sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

Network link 220 typically provides data communication through one or more networks to other data devices. For example, network link 220 may provide a connection through local network 222 to a host computer 224 or to data equipment operated by an Internet Service Provider (ISP) 226. ISP 226 in turn provides data communication services through the world wide packet data communication network now commonly referred to as the "Internet" 228. Local network 222 and Internet 228 both use electrical, electromagnetic or optical signals that carry digital data streams. The signals through the various networks and the signals on network link 220 and through communication interface 218, which carry the digital data to and from computer system 200, are exemplary forms of carrier waves transporting the information.

Computer system 200 can send messages and receive data, including program code, through the network(s), network link 220 and communication interface 218. In the Internet example, a server 230 might transmit a requested code for an application program through Internet 228, ISP 226, local network 222 and communication interface 218.

The received code may be executed by processor 204 as it is received, and/or stored in storage device 210, or other non-volatile storage for later execution. In this manner, computer system 200 may obtain application code in the form of a carrier wave.

In the foregoing specification, embodiments of the invention have been described with reference to numerous specific details that may vary from implementation to implementation. Thus, the sole and exclusive indicator of what is the invention, and is intended by the applicants to be the invention, is the set of claims that issue from this application, in the specific form in which such claims issue, including any subsequent correction. Any definitions set forth herein for terms contained in such claims shall govern the meaning of such terms as used in the claims. Hence, no limitation, element, property, feature, advantage or attribute that is not expressly recited in a claim should limit the scope of such claim in any way. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

Appendix I

- Introducing XML Schema
- Introducing XML Schema and Oracle XML DB
- Using Oracle XML DB and XML Schema
- Introducing DBMS_XMLSCHEMA
- Registering Your XML Schema Before Using Oracle XML DB
- Deleting Your XML Schema Using DBMS_XMLSCHEMA
- Guidelines for Using Registered XML Schemas
- Java Bean Generation During XML Schema Registration
- Generating XML Schema from Object-Relational Types Using DBMS_ XMLSCHEMA.generateSchema()
- XML Schema-Related Methods of XMLType
- Managing and Storing XML Schema
- DOM Fidelity
- Oracle XML DB Creates XMLType Tables and Columns Based on XML Schema
- Using SQLName and SQLType Attributes to Specify SQL Object Type Names Before Registering XML Schema
- Mapping of Types Using DBMS_XMLSCHEMA
- XML Schema: Mapping SimpleTypes to SQL
- XML Schema: Mapping ComplexTypes to SQL

- Oracle XML DB complexType Extensions and Restrictions
- Further Guidelines for Creating XML Schema-Based XML Tables
- Query Rewrite with XML Schema-Based Object-Relational Storage
- Creating Default Tables During XML Schema Registration
- Ordered Collections in Tables (OCTs)
- Cyclical References Between XML Schemas

Introducing XML Schema

The XML Schema recommendation was created by the World Wide Web Consortium (W3C) to describe the content and structure of XML documents in XML. It includes the full capabilities of Document Type Definitions (DTDs) so that existing DTDs can be converted to XML schema. XML schemas have additional capabilities compared to DTDs.

Introducing XML Schema and Oracle XML DB

XML Schema is a schema definition language written in XML. It can be used to describe the structure and various other semantics of conforming instance documents. For example, the following XML schema definition, *po.xsd*, describes the structure and other properties of purchase order XML documents.

This manual refers to XML schema definitions as *XML schema*.

Example 5–1 XML Schema Definition, po.xsd

```
-- The following is an example of an XML schema definition, po.xsd:
<schema targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd"
xmlns="http://www.w3.org/2001/XMLSchema">
 <complexType name="PurchaseOrderType">
  <sequence>
   <element name="PONum" type="decimal"/>
   <element name="Company">
    <simpleType>
     <restriction base="string">
      <maxLength value="100"/>
     </restriction>
    </simpleType>
   </element>
   <element name="Item" maxOccurs="1000">
    <complexType>
     <sequence>
      <element name="Part">
       <simpleType>
        <restriction base="string">
         <maxLength value="1000"/>
        </restriction>
       </simpleType>
```

```
      </element>
      <element name="Price" type="float"/>
     </sequence>
    </complexType>
   </element>
  </sequence>
 </complexType>
 <element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>
```

Example 5–2 XML Document, po.xml Conforming to XML Schema, po.xsd

```
-- The following is an example of an XML document that conforms to XML schema
po.xsd:
<PurchaseOrder xmlns="http://www.oracle.com/PO.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
     <PONum>1001</PONum>
     <Company>Oracle Corp</Company>
     <Item>
       <Part>9i Doc Set</Part>
       <Price>2550</Price>
     </Item>
</PurchaseOrder>
```

Note:

The URL `'http://www.oracle.com/PO.xsd'` used here is simply a name that uniquely identifies the registered XML schema within the database and need not be the physical URL at the which the XML schema document is located. Also, the target namespace of the XML schema is another URL, different from the XML schema location URL, that specifies an abstract namespace within which elements and types get declared.

An XML schema can optionally specify the target namespace URL. If this attribute is omitted, the XML schema has no target namespace. Note: The `targetnamespace` is commonly the same as XML schema's URL.

An XML instance document must specify both the namespace of the root element (same as the XML schema's target namespace) and the location (URL) of the XML schema that defines this root element. The location is specified with attribute `xsi:schemaLocation`. When the XML schema has no target namespace, use attribute `xsi:noNamespaceSchemaLocation` to specify the XML schema URL.

Using Oracle XML DB and XML Schema

Oracle XML DB uses annotated XML schema as metadata, that is, the standard XML Schema definitions along with several Oracle XML DB-defined attributes. These attributes are in a different namespace and control how instance documents get mapped into the database. Since these attributes are in a different namespace from the XML schema namespace, such annotated XML schemas are still legal XML schema documents:

When using Oracle XML DB, you must first register your XML schema. You can then use the XML schema URLs while creating `XMLType` tables, columns, and views.

Oracle XML DB provides XML schema support for the following tasks:

- Registering any W3C compliant XML Schema.

- Validating your XML documents against a registered XML schema definitions.
- Registering local and global XML schemas.
- Generating XML schema from object types.
- Referencing an XML schema owned by another user.
- Explicitly referencing a global XML schema when a local XML schema exists with the same name.
- Generating a structured database mapping from your XML schemas during XML schema registration. This includes generating SQL object types, collection types, and default tables, and capturing the mapping information using XML schema attributes.
- Specifying a particular SQL type mapping when there are multiple legal mappings.
- Creating `XMLType` tables, views and columns based on registered XML schemas.
- Performing manipulation (DML) and queries on XML schema-based `XMLType` tables.
- Automatically inserting data into default tables when schema-based XML instances are inserted into Oracle XML DB Repository using FTP, HTTP/WebDav protocols and other languages.

Why Do We Need XML Schema?

`XMLType` is a datatype that facilitates
storing XML in columns and tables in the database. XML schemas further facilitate storing XML columns and tables in the database, and they offer you more storage and access options for XML data along with space- performance-saving options.

For example, you can use XML schema to declare which elements and attributes can be used and what kinds of element nesting, and datatypes are allowed in the XML documents being stored or processed.

XML Schema Provide Flexible XML-to-SQL Mapping Set Up

Using XML schema with Oracle XML DB provides a flexible set up for XML storage mapping. For example:

- If your data is highly structured (mostly XML), each element in the XML documents can be stored as a column in a table.

- If your data is unstructured (all or mostly non-XML data), the data can be stored in a Character Large Object (CLOB).

Which storage method you choose depends on how your data will be used and depends on the queriability and your requirements for querying and updating your data. In other words. Using XML schema gives you more flexibility for storing highly structured or unstructured data.

XML Schema Allows XML Instance Validation

Another advantage for using XML schema with Oracle XML DB is that you can perform XML instance validation according to the XML schema and with respect to Oracle XML Repository requirements for optimal performance. For example,an XML schema can check that all incoming XML documents comply with definitions declared in the XML schema, such as allowed structure, type, number of allowed item occurences, or allowed length of items.

Also, by registering XML schema in Oracle XML DB, when inserting and storing XML instances using Protocols, such as FTP or HTTP, the XML schema information can influence how efficiently XML instances are inserted.

When XML instances must be handled without any prior information about them, XML schema can be useful in predicting optimum storage, fidelity, and access.

Introducing DBMS_XMLSCHEMA

Oracle XML DB's XML schema functionality is available through the PL/SQL supplied package, `DBMS_XMLSCHEMA`, a server-side component that handles the registration of XML schema definitions for use by Oracle XML DB applications.

Two of the main `DBMS_XMLSCHEMA` functions are:

- **`registerSchema()`**. This registers an XML schema given:
  - XML schema source, which can be in a variety of formats, including string, LOB, XMLType, and URIType
  - Its schema URL or XMLSchema name
- **`deleteSchema()`**. This deletes a previously registered XML schema, identified by its URL or XMLSchema name.

Registering Your XML Schema Before Using Oracle XML DB

An XML schema must be registered before it can be used or referenced in any context by Oracle XML DB. XML schema are registered by using `DBMS_XMLSCHEMA.registerSchema()` and specifying the following:

- The XML schema source document as a VARCHAR, CLOB, `XMLType`, or URIType.
- The XML schema URL. This is a name for the XML schema that is used within XML instance documents to specify the location of the XML schema to which they conform.

After registration has completed:

- XML documents conforming to this XML schema, and referencing it using the XML schema's URL within the XML document, can be processed by Oracle XML DB.
- Tables and columns can be created for root XML elements defined by this XML schema to store the conforming XML documents.

Registering Your XML Schema using DBMS_XMLSCHEMA

Use `DBMS_XMLSCHEMA` to register your XML schema. This involves specifying the XML schema document and its URL, also known as the *XML schema location*.

Example 5–3 Registering an XML Schema That Declares a complexType Using DBMS_XMLSCHEMA

```
-- Consider the XML schema shown below. It declares a complexType called
PurchaseOrderType
-- and an element PurchaseOrder of this type. The schema is stored in the
-- PL/SQL variable doc. The following registers the XML schema at URL:
-- http://www.oracle.com/PO.xsd.

declare
        doc varchar2(1000) := '<schema
targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd"
xmlns="http://www.w3.org/2001/XMLSchema">
 <complexType name="PurchaseOrderType">
  <sequence>
   <element name="PONum" type="decimal"/>
   <element name="Company">
    <simpleType>
```

```
      <restriction base="string">
       <maxLength value="100"/>
      </restriction>
     </simpleType>
    </element>
    <element name="Item" maxOccurs="1000">
     <complexType>
      <sequence>
       <element name="Part">
        <simpleType>
         <restriction base="string">
          <maxLength value="1000"/>
         </restriction>
        </simpleType>
       </element>
       <element name="Price" type="float"/>
      </sequence>
     </complexType>
    </element>
   </sequence>
  </complexType>
  <element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>';
begin
        dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

-- The registered schema can be used to created XMLSchema-Based tables, or
-- XMLSchema-based columns. For example, the following statement creates an
-- a table with an XMLSchema-based column.
create table po_tab(
 id number,
 po sys.XMLType
)
  xmltype column po
    XMLSCHEMA "http://www.oracle.com/PO.xsd"
    element "PurchaseOrder";

-- The following shows an XMLType instance that conforms to the preceding XML
-- schema being inserted into the above table. The schemaLocation attribute
-- specifies the schema URL:

insert into po_tab values (1,
xmltype('<PurchaseOrder xmlns="http://www.oracle.com/PO.xsd"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
 xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
     <PONum>1001</PONum>
     <Company>Oracle Corp</Company>
     <Item>
       <Part>9i Doc Set</Part>
       <Price>2550</Price>
     </Item>
     <Item>
       <Part>8i Doc Set</Part>
       <Price>350</Price>
     </Item>
</PurchaseOrder>'));
```

Local and Global XML Schemas

XML schemas can be registered as local or global:

- *Local XML schema*: An XML schema registered as a local schema is, by default, visible only to the owner.
- *Global XML schema*: An XML schema registered as a global schema is, by default, visible and usable by all database users.

When you register an XML schema, `DBMS_XMLSCHEMA` adds an Oracle XML DB resource corresponding to the XML schema into the Oracle XML DB Repository. The XML schema URL determines the path name of the resource in Oracle XML DB Repository according to the following rules:

Local XML Schema

In Oracle XML DB, *local XML schema* resources are created under `/sys/schemas/<username>` directory. The rest of the path name is derived from the schema URL.

Example 5–4 A Local XML Schema

For example, a local XML schema with schema URL:

```
http://www.myco.com/PO.xsd
```

registered by SCOTT, is given the path name:

```
/sys/schemas/SCOTT/www.myco.com/PO.xsd.
```

Database users need appropriate permissions (ACLs) to create a resource with this path name in order to register the XML schema as a local XML schema.

By default, an XML schema belongs to you after registering the XML schema with Oracle XML DB. A reference to the XML schema document is stored in Oracle XML DB Repository, in directory:

```
/sys/schemas/<username>/....
```

For example, if you, SCOTT, registered the preceding XML schema, it is mapped to the file:

```
/sys/schemas/SCOTT/www.oracle.com/PO.xsd
```

Such XML schemas are referred to as *local*. In general, they are usable only by you to whom they belong.

Note: Typically, only the *owner* of the XML schema can use it to define `XMLType` tables, columns, or views, validate documents, and so on. However, Oracle supports fully qualified XML schema URLs which can be specified as:

```
http://xmlns.oracle.com/xdb/schemas/SCOTT/www.orac
le.com/PO.xsd
```

This extended URL can be used by privileged users to specify XML schema belonging to other users.

Global XML Schema

In contrast to local schema, privileged users can register an XML schema as a *global XML schema* by specifying an argument in the `DBMS_XMLSCHEMA` registration function.

Global schemas are visible to *all* users and stored under the `/sys/schemas/PUBLIC/` directory in Oracle XML DB Repository.

Note: Access to this directory is controlled by Access Control Lists (ACLs) and, by default, is writeable only by a DBA. You need WRITE privileges on this directory to register global schemas.

`XDBAdmin` role also provides WRITE access to this directory, assuming that it is protected by the default "protected" ACL.

You can register a local schema with the same URL as an existing global schema. A local schema always hides any global schema with the same name (URL).

Example 5–5 A Global XML Schema

For example, a global schema registered by SCOTT with the URL:

```
www.myco.com/PO.xsd
```

is mapped to Oracle XML DB Repository at:

```
/sys/schemas/PUBLIC/www.myco.com/PO.xsd
```

Database users need appropriate permissions (ACLs) to create this resource in order to register the XML schema as *global*.

Registering Your XML Schema: Oracle XML DB Sets Up the Storage and Access Infrastructure

As part of registering an XML schema, Oracle XML DB also performs several other steps to facilitate storing, accessing, and manipulating XML instances that conform to the XML schema. These steps include:

- *Creating types:* When an XML schema is registered, Oracle creates the appropriate SQL object types that enable the structured storage of XML documents that conform to this XML schema. You can use Oracle XML DB-defined attributes in XML schema documents to control how these object types are generated.

- *Creating default tables:* As part of XML schema registration, Oracle XML DB generates default `XMLType` tables for all root elements. You can also specify any column and table level constraints for use during table creation.

- *Creating Java beans:* Java beans can be optionally generated during XML schema registration. These Java classes provide accessor and mutator methods for elements and attributes declared in the schema. Access using Java beans offers better performance for manipulating XML when the XML schema is well known. This helps avoid run-time name translation.

Deleting Your XML Schema Using DBMS_XMLSCHEMA

You can delete your registered XML schema by using the `DBMS_XMLSCHEMA.deleteSchema` procedure. When you attempt to delete an XML schema, `DBMS_XMLSCHEMA` checks:

- That the current user has the appropriate privileges (ACLs) to delete the resource corresponding to the XML schema within Oracle XML DB Repository. You can thus control which users can delete which XML schemas by setting the appropriate ACLs on the XML schema resources.
- For dependents. If there are any dependents, it raises an error and the deletion operation fails. This is referred to as the **RESTRICT** mode of deleting XML schemas.

FORCE Mode

A *FORCE mode* option is provided while deleting XML schemas. If you specify the FORCE mode option, the XML schema deletion proceeds even if it fails the dependency check. In this mode, XML schema deletion marks all its dependents as invalid.

CASCADE Mode

The *CASCADE mode* option drops all generated types, default tables, and Java beans as part of a previous call to register schema.

> **See Also:** *Oracle9i XML API Reference - XDK and XDB* chapter on DBMS_XMLSCHEMA

Example 5–6 Deleting the XML Schema Using DBMS_XMLSCHEMA

```
-- The following example deletes XML schema PO.xsd. First, the dependent table
-- po_tab is dropped. Then, the schema is deleted using the FORCE and CASCADE
-- modes with DBMS_XMLSCHEMA.DELETESCHEMA.

drop table po_tab;

EXEC dbms_xmlschema.deleteSchema('http://www.oracle.com/PO.xsd',
            dbms_xmlschema.DELETE_CASCADE_FORCE);
```

Guidelines for Using Registered XML Schemas

The following sections describe guidelines for registering XML schema with Oracle XML DB.

Objects That Depend on Registered XML Schemas

The following objects depend on a registered XML schemas:

- Tables or views that have an XMLType column that conforms to some element in the XML schema.
- XML schemas that include or import this schema as part of their definition.
- Cursors that reference the XML schema name, for example, within DBMS_XMLGEN operators. Note that these are purely transient objects.

Creating XMLType Tables, Views, or Columns

After an XML schema has been registered, it can be used to create XML schema-based XMLType tables, views, and columns by referencing the following:

- The XML schema URL of a registered XML schema
- The name of the root element

Example 5–7 Post-Registration Creation of an XMLType Table

```
-- For example you can create an XMLSchema-based XMLType table as follows:
CREATE TABLE po_tab OF XMLTYPE
      XMLSCHEMA "http://www.oracle.com/PO.xsd" ELEMENT "PurchaseOrder";

-- The following statement inserts schema-conformant data.
insert into po_tab values (
  xmltype('<PurchaseOrder xmlns="http://www.oracle.com/PO.xsd"
```

```
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
     <PONum>1001</PONum>
     <Company>Oracle Corp</Company>
     <Item>
       <Part>9i Doc Set</Part>
       <Price>2550</Price>
     </Item>
     <Item>
       <Part>8i Doc Set</Part>
       <Price>350</Price>
     </Item>
</PurchaseOrder>'));
```

Validating XML Instances Against the XML Schema: schemaValidate()

You can validate an XMLType instance against a registered XML schema by using one of the validation methods.

Example 5–8 Validating XML Using schemaValidate()

```
-- The following PL/SQL example validates an XML instance against XML schema
PO.xsd:
declare
  xmldoc xmltype;
  begin

  -- populate xmldoc (by fetching from table)
  select value(p) into xmldoc from po_tab p;

  --   validate against XML schema
  xmldoc.schemavalidate();

  if xmldoc.isschemavalidated() = 1 then
                    dbms_output.put_line('Data is valid');
  else
                    dbms_output.put_line('Data is invalid');
  end if;
end;
```

Fully Qualified XML Schema URLs

By default, XML schema URL names are always referenced within the scope of the current user. In other words, when database users specify XML schema URLs, they are first resolved as the names of local XML schema owned by the current user.

- If there are no such XML schemas, then they are resolved as names of *global* XML schema.
- If there are no *global* XML schema, then Oracle XML DB raises an error.

XML Schema That Users Cannot Reference

These rules imply that, by default, users cannot reference the following kinds of XML schemas:

- XML schemas owned by a different database user
- Global XML schemas that have the same name as local XML schemas

Fully Qualified XML Schema URLs Permit Explicit Reference to XML Schema URLs

To permit explicit reference to XML schemas in these cases, Oracle XML DB supports a notion of *fully qualified* XML schema URLs. In this form, the name of the database user owning the XML schema is also specified as part of the XML schema URL, except that such XML schema URLs belong to the Oracle XML DB namespace as follows:

```
http://xmlns.oracle.com/xdb/schemas/<database-user-name>/<schemaURL-
minus-protocol>
```

Example 5–9 Using Fully Qualified XML Schema URL

```
-- For example, consider the global XML schema with the following URL:
http://www.example.com/po.xsd

-- Assume that database user SCOTT has a local XML schema with the same URL:
http://www.example.com/po.xsd

-- User JOE can reference the local XML schema owned by SCOTT as follows:
http://xmlns.oracle.com/xdb/schemas/SCOTT/www.example.com/po.xsd

-- Similarly, the fully qualified URL for the global XML schema is:
http://xmlns.oracle.com/xdb/schemas/PUBLIC/www.example.com/po.xsd
```

Transactional Behavior of XML Schema Registration

Registration of an XML schema is non transactional and auto committed as with other SQL DDL operations, as follows:

- If registration succeeds, the operation is auto committed.
- If registration fails, the database is rolled back to the state before the registration began.

Since XML schema registration potentially involves creating object types and tables, error recovery involves dropping any such created types and tables. Thus, the entire XML schema registration is guaranteed to be atomic. That is, either it succeeds or the database is restored to the state before the start of registration.

Java Bean Generation During XML Schema Registration

Java beans can be optionally generated during XML schema registration and provide accessor and mutator methods for elements and attributes declared in the XML schema. Access to XML data stored in the database, using Java beans, offers better performance for manipulating the XML data when the XML schema is well known and mostly fixed by avoiding run-time name translation.

Example 5–10 Generating Java Bean Classes During XMl Schema Registration

For example, the Java bean class corresponding to the XML schema `PO.xsd` has the following accessor and mutator methods:

```
public class PurchaseOrder extends XMLTypeBean
{
  public BigDecimal getPONum()
  {
     ....
  }
  public void setPONum(BigDecimal val)
  {
     ....
  }
  public String getCompany()
  {
     ....
  }
  public void setCompany(String val)
  {
     ....
```

```
  }
....
}
```

Note: Java Bean support in Oracle XML DB is only for XML schema-based XML documents. Non-schema-based XML documents can be manipulated using Oracle XML DB DOM API.

Generating XML Schema from Object-Relational Types Using DBMS_XMLSCHEMA.generateSchema()

An XML schema can be generated from an object-relational type automatically using a default mapping. The `generateSchema()` and `generateSchemas()` functions in the `DBMS_XMLSCHEMA` package take in a string that has the object type name and another that has the Oracle XML DB XMLschema.

- `generateSchema()` returns an `XMLType` containing an XML schema. It can optionally generate XML schema for all types referenced by the given object type or restricted only to the top-level types.
- `generateSchemas()` is similar, except that it returns an `XMLSequenceType` of XML schemas, each corresponding to a different namespace. It also takes an additional optional argument, specifying the root URL of the preferred XML schema location:

  `http://xmlns.oracle.com/xdb/schemas/<schema>.xsd`

  They can also optionally generate annotated XML schemas that can be used to register the XML schema with Oracle XML DB.

Example 5–11 Generating XML Schema: Using generateSchema()

```
-- For example, given the object type:
connect tl/tl
CREATE TYPE employee_t AS OBJECT
(
   empno NUMBER(10),
   ename VARCHAR2(200),
   salary NUMBER(10,2)
);

-- We can generate the schema for this type as follows :
```

```
select  dbms_xmlschema.generateschema('T1', 'EMPLOYEE_T') from dual;

-- This returns a schema corresponding to the type EMPLOYEE_T. The schema
-- declares an ELEMENT named EMPLOYEE_T and a COMPLEXTYPE called EMPLOYEE_TType.
-- The schema includes other annotation from http://xmlns.oracle.com/xdb.

DBMS_XMLSCHEMA.GENERATESCHEMA('T1','EMPLOYEE_T')
--------------------------------------------------------------------------------
<xsd:schema targetNamespace="http://ns.oracle.com/xdb/T1" xmlns="http://ns.oracl
e.com/xdb/T1" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xdb="http://xml
ns.oracle.com/xdb" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:sch
emaLocation="http://xmlns.oracle.com/xdb http://xmlns.oracle.com/xdb/XDBSchema.x
sd">
  <xsd:element name="EMPLOYEE_T" type="EMPLOYEE_TType" xdb:SQLType="EMPLOYEE_T"
xdb:SQLSchema="T1"/>
  <xsd:complexType name="EMPLOYEE_TType">
    <xsd:sequence>
      <xsd:element name="EMPNO" type="xsd:double" xdb:SQLName="EMPNO" xdb:SQLTyp
e="NUMBER"/>
      <xsd:element name="ENAME" type="xsd:string" xdb:SQLName="ENAME" xdb:SQLTyp
e="VARCHAR2"/>
      <xsd:element name="SALARY" type="xsd:double" xdb:SQLName="SALARY" xdb:SQLT
ype="NUMBER"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

XML Schema-Related Methods of XMLType

Table 5–1 lists the XMLType API's XML schema-related methods.

Table 5–1 XMLType API XML Schema-Related Methods

XMLType API Method	Description
isSchemaBased()	Returns TRUE if the XMLType instance is based on an XML schema, FALSE otherwise.
getSchemaURL() getRootElement() getNamespace()	Returns the XML schema URL, name of root element, and the namespace for an XML schema-based XMLType instance.
schemaValidate() isSchemaValid() is SchemaValidated() setSchemaValidated()	An XMLType instance can be validated against a registered XML schema using the validation methods.

Managing and Storing XML Schema

XML schema documents are themselves stored in Oracle XML DB as XMLType instances. XML schema-related XMLType types and tables are created as part of the Oracle XML DB installation script, catxdbs.sql.

Root XML Schema, XDBSchema.xsd

The XML schema for XML schemas, is called the root XML schema, XDBSchema.xsd. XDBSchema.xsd describes any valid XML schema document that can be registered by Oracle XML DB. You can access XDBSchema.xsd through Oracle XML DB Repository at:

/sys/schemas/PUBLIC/xmlns.oracle.com/xdb/XDBSchema.xsd

How Are XML Schema-Based XMLType Structures Stored?

XML Schema-based XMLType structures are stored in one of the following ways:

- *In underlying object type columns.* This is the default storage mechanism.
  - SQL object types can be created optionally during the XML schema registration process.
  - Mappings from XML to SQL object types and attributes, is stored in the XML schema document as extra annotations, that is, as attributes defined by Oracle XML DB and defined in http://xmlns.oracle.com/xdb.
- *In a single underlying LOB column.* Here the storage choice is specified in the STORE AS clause of the CREATE TABLE statement:

```
CREATE TABLE po_tab OF xmltype
    STORE AS CLOB
    ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

Specifying the Storage Mechanism

Instead of using the STORE AS clause, you can specify that the table and column be stored according to a mapping based on a particular XML schema. You can specify the URL for the XML schema used for the mapping.

Non-XML schema-based XML data can be stored in tables using CLOBs. However you do not obtain benefits such as indexing, query-rewrite, and so on.

DOM Fidelity

Document Object Model (DOM) fidelity is the concept of retaining the structure of a retrieved XML document, compared to the original XML document, for DOM traversals. DOM fidelity is needed to ensure the accuracy and integrity of XML documents stored in Oracle XML DB.

How Oracle XML DB Ensures DOM Fidelity with XML Schema

All elements and attributes declared in the XML schema are mapped to separate attributes in the corresponding SQL object type. However, some pieces of information in XML instance documents are not represented directly by these element or attributes, such as:

- Comments
- Namespace declarations
- Prefix information

To ensure the integrity and accuracy of this data, for example, when regenerating XML documents stored in the database, Oracle XML DB uses a data integrity mechanism called *DOM fidelity.*

DOM fidelity refers to how identical the *returned* XML documents are compared to the *original* XML documents, particularly for purposes of DOM traversals.

DOM Fidelity and SYS_XDBPD$

To guarantee that DOM fidelity is maintained and that the *returned* XML documents are identical to the *original* XML document for DOM traversals, Oracle XML DB adds a system binary attribute, `SYS_XDBPD$`, to each created object type.

This attribute stores all pieces of information that cannot be stored in any of the other attributes, thereby ensuring the DOM fidelity of all XML documents stored in Oracle XML DB. Examples of such pieces of information include: ordering information, comments, processing instructions, namespace prefices, and so on.

This is mapped to a Positional Descriptor (PD) column.

Note: In general,it is not a good idea to set this information because the extra pieces of information, such as, comments, processing instructions, and so on, could be lost if there is no PD column.

How to Suppress SYS_XDBPD$

If DOM fidelity is not required, you can suppress `SYS_XDBPD$` in the XML schema definition by setting the attribute, `maintainDOM=FALSE`.

Note: The attribute SYS_XDBPD$ is omitted in many examples here for clarity. However, the attribute is always present as a Positional Descriptor (PD) column in all SQL object types generated by the XML schema registration process.

Oracle XML DB Creates XMLType Tables and Columns Based on XML Schema

Oracle XML DB creates XML Schema-based XMLType tables and columns by referencing:

- The XML schema URL of a registered XML schema
- The name of the root element

Figure 3 shows the syntax for creating an XMLType table:

```
CREATE TABLE [schema.] table OF XMLTYPE
  [XMLTYPE XMLType_storage] [XMLSchema_spec];
```

A subset of the XPointer notation, shown in the following example, can also be used to provide a single URL containing the XML schema location and element name.

Example 5–12 Creating XML Schema-Based XMLType Table

This example creates the XMLType table po_tab using the XML schema at the given URL:

```
CREATE TABLE po_tab OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/PO.xsd" ELEMENT "PurchaseOrder";
```

An equivalent definition is:

```
CREATE TABLE po_tab OF XMLTYPE
    ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

SQL Object-Relational Types Store XML Schema-Based XMLType Tables

When an XML schema is registered, Oracle XML DB creates the appropriate SQL object types that enable structured storage of XML documents that conform to this XML schema. All SQL object types are created based on the current registered XML schema, by default.

Example 5–13 Creating SQL Object Types to Store XMLType Tables

```
-- For example, when PO.xsd is registered with Oracle XML DB, the following SQL
types
-- are created. Note that the names of the types are generated names, and will
not
-- nescessarily match Itemxxx_t, Itemxxx_COLL and PurchaseOrderTypexxx_T, where
xxx
-- is a 3-digit integer.
CREATE TYPE "Itemxxx_T" as object
(
    part varchar2(1000),
    price number
);

CREATE TYPE "Itemxxx_COLL" AS varray(1000) OF "Item_T";
CREATE TYPE "PurchaseOrderTypexxx_T" AS OBJECT
(
 ponum number,
 company varchar2(100),
 item Item_varray_COLL
);
```

Note: The names of the object types and attributes in the preceding example can be system-generated.

- If the XML schema already contains the `SQLName`, `SQLType`, or `SQLColType` attribute filled in

  , this name
  is used as the object attribute's name.

- If the XML schema does not contain the `SQLName` attribute, the name is derived from the XML name, unless it cannot be used because of length or conflict reasons.

If the `SQLSchema` attribute is used, Oracle XML DB attempts to create the object type using the specified database schema. The current user must have the necessary privileges to perform this.

Using SQLName and SQLType Attributes to Specify SQL Object Type Names Before Registering XML Schema

To specify specific names of SQL objects generated include the attributes `SQLName` and `SQLType` in the XML schema definition prior to registering the XML schema.

- If you specify the `SQLName` and `SQLType` values, Oracle XML DB creates the SQL object types using these names.
- If you do not specify these attributes, Oracle XML DB uses system-generated names.

Note: You do not have to specify values for any of these attributes. Oracle XML DB fills in appropriate values during the XML schema registration process. However, it is recommended that you specify the names of at least the top-level SQL types so that you can reference them later.

All annotations are in the form of attributes that can be specified within attribute and element declarations. These attributes belong to the Oracle XML DB namespace: `http://xmlns.oracle.com/xdb`

Table 5–2 lists Oracle XML DB attributes that you can specify in element and attribute declarations.

Table 5–2 AttributesYou Can Specify in Elements

Attribute	Values	Default	Description
SQLName	Any SQL identifier	Element name	Specifies the name of the attribute within the SQL object that maps to this XML element.
SQLType	Any SQL type name	Name generated from element name	Specifies the name of the SQL type corresponding to this XML element declaration.
SQLCollType	Any SQL collection type name	Name generated from element name	Specifies the name of the SQL collection type corresponding to this XML element that has `maxOccurs` > 1.
SQLSchema	Any SQL username	User registering XML schema	Name of database user owning the type specified by SQLType.
SQLCollSchema	Any SQL username	User registering XML schema	Name of database user owning the type specified by `SQLCollType`.
maintainOrder	true | false	true	If true, the collection is mapped to a VARRAY. If false, the collection is mapped to a NESTED TABLE.
SQLInline	true | false	true	If true this element is stored inline as an embedded attribute (or a collection if `maxOccurs` > 1). If false, a REF (or collection of REFs if `maxOccurs` > 1) is stored. This attribute will be forced to false in certain situations (like cyclic references) where SQL will not support inlining.
maintainDOM	true | false	true	If true, instances of this element are stored such that they retain DOM fidelity on output. This implies that all comments, processing instructions, namespace declarations, and so on are retained in addition to the ordering of elements. If false, the output need not be guaranteed to have the same DOM behavior as the input.

Table 5–2 AttributesYou Can Specify in Elements(Cont.)

Attribute	Values	Default	Description
columnProps	Any valid column storage clause	NULL	Specifies the column storage clause that is inserted into the default CREATE TABLE statement. It is useful mainly for elements that get mapped to tables , namely top-level element declarations and out-of-line element declarations.
tableProps	Any valid table storage clause	NULL	Specifies the TABLE storage clause that is appended to the default CREATE TABLE statement. This is meaningful mainly for global and out-of-line elements.
defaultTable	Any table name	Based on element name.	Specifies the name of the table into which XML instances of this schema should be stored. This is most useful in cases when the XML is being inserted from APIs where table name is not specified, for example, FTP and HTTP.
beanClassname	Any Java class name	Generated from element name.	Can be used within element declarations. If the element is based on a global `complexType`, this name must be identical to the `beanClassname` value within the `complexType` declaration. If a name is specified by the user, the bean generation will generate a bean class with this name instead of generating a name from the element name.
JavaClassname	Any Java class name	None	Used to specify the name of a Java class that is derived from the corresponding bean class to ensure that an object of this class is instantiated during bean access. If a `JavaClassname` is not specified, Oracle XML DB will instantiate an object of the bean class directly.

Table 5–3 Attributes You Can Specify in Elements Declaring Global complexTypes

Attribute	Values	Default	Description
SQLType	Any SQL type name	Name generated from element name	Specifies the name of the SQL type corresponding to this XML element declaration.
SQLSchema	Any SQL username	User registering XML schema	Name of database user owning the type specified by `SQLType`.
beanClassname	Any Java class name	Generated from element name.	Can be used within element declarations. If the element is based on a global `complexType`, this name must be identical to the `beanClassname` value within the `complexType` declaration. If a name is specified by the user, the bean generation will generate a bean class with this name, instead of generating a name from the element name.
maintainDOM	true | false	true	If true, instances of this element are stored such that they retain DOM fidelity on output. This implies that all comments, processing instructions, namespace declarations, and so on, are retained in addition to the ordering of elements. If false, the output need not be guaranteed to have the same DOM behavior as the input.

Table 5–4 Attributes You Can Specify in XML Schema Declarations

Attribute	Values	Default	Description
mapUnboundedStringToLob	true | false	false	If true, unbounded strings are mapped to CLOB by default. Similarly, unbounded binary data gets mapped to BLOB, by default. If false, unbounded strings are mapped to VARCHAR2(4000) and unbounded binary components are mapped to RAW(2000).
storeVarrayAsTable	true | false	false	If true, the VARRAY is stored as a table (OCT). If false, the VARRAY is stored in a LOB.

SQL Mapping Is Specified in the XML Schema During Registration

Information regarding the SQL mapping is stored in the XML schema document. The registration process generates the SQL types, and adds annotations to the XML schema document to store the mapping information. Annotations are in the form of new attributes.

Example 5–14 Capturing SQL Mapping Using SQLType and SQLName Attributes

```
-- The following XML schema definition shows how SQL mapping information
-- is captured using SQLType and SQLName attributes:

declare
     doc varchar2(3000) := '<schema
targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd" xmlns:xdb="http://xmlns.oracle.com/xdb"
xmlns="http://www.w3.org/2001/XMLSchema">
 <complexType name="PurchaseOrderType">
  <sequence>
   <element name="PONum" type="decimal" xdb:SQLName="PONUM"
xdb:SQLType="NUMBER"/>
   <element name="Company" xdb:SQLName="COMPANY" xdb:SQLType="VARCHAR2">
    <simpleType>
     <restriction base="string">
      <maxLength value="100"/>
     </restriction>
    </simpleType>
   </element>
   <element name="Item"  xdb:SQLName="ITEM" xdb:SQLType="ITEM_T"
maxOccurs="1000">
    <complexType>
     <sequence>
      <element name="Part"  xdb:SQLName="PART" xdb:SQLType="VARCHAR2">
       <simpleType>
        <restriction base="string">
         <maxLength value="1000"/>
        </restriction>
       </simpleType>
      </element>
      <element name="Price" type="float"  xdb:SQLName="PRICE"
xdb:SQLType="NUMBER"/>
     </sequence>
    </complexType>
   </element>
```

```
  </sequence>
 </complexType>
 <element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>';
begin
   dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;
```

Figure 4 shows how Oracle XML DB creates XML schema-based XMLType tables using an XML document and mapping specified in an XML schema. An XMLType table is first created and depending on how the storage is specified in the XML schema, the XMl document is mapped and stored either as a CLOB in one XMLType column, or stored object-relationally and spread out across several columns in the table.

An XMLType table is first created and depending on how the storage is specified in the XML schema, the XMl document is mapped and stored either as a CLOB in one XMLType column, or stored object-relationally and spread out across several columns in the table.

Mapping of Types Using DBMS_XMLSCHEMA

Use DBMS_XMLSCHEMA to set the mapping of type information for attributes and elements.

Setting Attribute Mapping Type Information

An attribute declaration can have its type specified in terms of one of the following:

- Primitive type
- Global `simpleType`, declared within this XML schema or in an external XML schema
- Reference to global attribute (`ref=" . . "`), declared within this XML schema or in an external XML schema
- Local `simpleType`

In all cases, the SQL type and associated information (length and precision) as well as the memory mapping information, are derived from the `simpleType` on which the attribute is based.

Overriding SQL Types

You can explicitly specify an `SQLType` value in the input XML schema document. In this case, your specified type is validated. This allows for the following specific forms of overrides:

- If the default type is a STRING, you can override it with any of the following: CHAR, VARCHAR, or CLOB.
- If the default type is RAW, you can override it with RAW or BLOB.

Setting Element Mapping Type Information

An element declaration can specify its type in terms of one of the following:

- Any of the ways for specifying type for an attribute declaration.
- Global `complexType`, specified within this XML schema document or in an external XML schema.
- Reference to a global element (`ref=" . . . "`), which could itself be within this XML schema document or in an external XML schema.

- Local `complexType`.

Overriding SQL Type

An element based on a `complexType` is, by default, mapped to an object type containing attributes corresponding to each of the sub-elements and attributes. However, you can override this mapping by explicitly specifying a value for `SQLType` attribute in the input XML schema. The following values for `SQLType` are permitted in this case:

- VARCHAR2
- RAW
- CLOB
- BLOB

These represent storage of the XML in a text or unexploded form in the database. The following special cases are handled:

- If a cycle is detected, as part of processing the `complexTypes` used to declare elements and elements declared within the `omplexType`), the `SQLInline` attribute is forced to be "false" and the correct SQL mapping is set to REF `XMLTYPE`.
- If `maxOccurs > 1`, a VARRAY type may need to be created.
  - If `SQLInline ="true"`, a varray type is created whose element type is the SQL type previously determined.
    * Cardinality of the VARRAY is determined based on the value of `maxOccurs` attribute.
    * The name of the VARRAY type is either explicitly specified by the user using `SQLCollType` attribute or obtained by mangling the element name.
  - If `SQLInline="false"`, the SQL type is set to `XDB.XDB$XMLTYPE_REF_LIST_T`, a predefined type representing an array of REFs to `XMLType`.
- If the element is a global element, or if `SQLInline="false"`, a default table needs to be created. It is added to the table creation context. The name of the default table has either been specified by the user, or derived by mangling the element name.

XML Schema: Mapping SimpleTypes to SQL

This section describes how XML schema definitions are used to map XML schema `simpleType` to SQL object types.

Table 5–5 through Table 5–8 list the default mapping of XML schema `simpleType` to SQL, as specified in the XML schema definition. For example:

- An XML primitive type is mapped to the closest SQL datatype. For example, DECIMAL, POSITIVEINTEGER, and FLOAT are all mapped to SQL NUMBER.
- An XML enumeration type is mapped to an object type with a single RAW(n) attribute. The value of n is determined by the number of possible values in the enumeration declaration.
- An XML list or a union datatype is mapped to a string (VARCHAR2/CLOB) datatype in SQL.

Table 5–5 Mapping XML String Datatypes to SQL

XML Primitive Type	Length or MaxLength Facet	Default Mapping	Compatible Datatype
string	n	VARCHAR2(n) if n < 4000, else VARCHAR2(4000)	CHAR,VARCHAR2, CLOB
string	--	VARCHAR2(4000) if mapUnboundedStringToLob="true", CLOB	CHAR, VARCHAR2, CLOB

Table 5–6 Mapping XML Binary Datatypes (hexBinary/base64Binary) to SQL

XML Primitive Type	Length or MaxLength Facet	Default Mapping	Compatible Datatypes
hexBinary, base64Binary	n	RAW(n) if n < 2000, else RAW(2000)	RAW, BLOB
hexBinary, base64Binary	-	RAW(2000) if mapUnboundedStringToLob="true", BLOB	RAW, BLOB

Table 5–7 Default Mapping of Numeric XML Primitive Types to SQL

XML Simple Type	Default Oracle DataType	totalDigits (m), fractionDigits(n) Specified	Compatible Datatypes
float	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
double	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
decimal	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
integer	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
nonNegativeInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
positiveInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
nonPositiveInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
negativeInteger	NUMBER	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
long	NUMBER(20)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedLong	NUMBER(20)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
int	NUMBER(10)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedInt	NUMBER(10)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
short	NUMBER(5)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedShort	NUMBER(5)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
byte	NUMBER(3)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE
unsignedByte	NUMBER(3)	NUMBER(m,n)	NUMBER, FLOAT, DOUBLE

Table 5–8 Mapping XML Date Datatypes to SQL

XML Primitive Type	Default Mapping	Compatible Datatypes
datetime	TIMESTAMP	DATE
time	TIMESTAMP	DATE
date	DATE	DATE
gDay	DATE	DATE
gMonth	DATE	DATE
gYear	DATE	DATE

Table 5–8 Mapping XML Date Datatypes to SQL (Cont.)

XML Primitive Type	Default Mapping	Compatible Datatypes
gYearMonth	DATE	DATE
gMonthDay	DATE	DATE
duration	VARCHAR2(4000)	none

Table 5–9 Default Mapping of Other XML Primitive Datatypes to SQL

XML Simple Type	Default Oracle DataType	Compatible Datatypes
boolean	RAW(1)	VARCHAR2
Language(string)	VARCHAR2(4000)	CLOB, CHAR
NMTOKEN(string)	VARCHAR2(4000)	CLOB, CHAR
NMTOKENS(string)	VARCHAR2(4000)	CLOB, CHAR
Name(string)	VARCHAR2(4000)	CLOB, CHAR
NCName(string)	VARCHAR2(4000)	CLOB, CHAR
ID	VARCHAR2(4000)	CLOB, CHAR
IDREF	VARCHAR2(4000)	CLOB, CHAR
IDREFS	VARCHAR2(4000)	CLOB, CHAR
ENTITY	VARCHAR2(4000)	CLOB, CHAR
ENTITIES	VARCHAR2(4000)	CLOB, CHAR
NOTATION	VARCHAR2(4000)	CLOB, CHAR
anyURI	VARCHAR2(4000)	CLOB, CHAR
anyType	VARCHAR2(4000)	CLOB, CHAR
anySimpleType	VARCHAR2(4000)	CLOB, CHAR
QName	XDB.XDB$QNAME	--

simpleType: Mapping XML Strings to SQL VARCHAR2 Versus CLOBs

If the XML schema specifies the datatype to be string with a `maxLength` value of less than 4000, it is mapped to a VARCHAR2 attribute of the specified length. However, if `maxLength` is not specified in the XML schema, it can only be mapped to a LOB. This is sub-optimal when most of the string values are small and only a small fraction of them are large enough to need a LOB. See Figure 5.

Figure 5–3 Oracle XML DB: Mapping XML Strings to SQL VARCHAR2 or CLOBs

XML Schema: Mapping ComplexTypes to SQL

Using XML schema, a `complexType` is mapped to an SQL object type as follows:

- *XML attributes* declared within the `complexType` are mapped to *object attributes*. The `simpleType` defining the XML attribute determines the SQL datatype of the corresponding attribute.
- *XML elements* declared within the `complexType` are also mapped to *object attributes*. The datatype of the object attribute is determined by the `simpleType` or `complexType` defining the XML element.

If the XML element is declared with attribute `maxOccurs` > 1, it is mapped to a `collection` attribute in SQL. The `collection` could be a VARRAY (default) or nested table if the `maintainOrder` attribute is set to false. Further, the default storage of the VARRAY is in Ordered Collections in Tables (OCTs) instead of LOBs. You can choose LOB storage by setting the `storeAsLob` attribute to true.

See Also: "Ordered Collections in Tables (OCTs)" on page 5-70

Mapping complexType to SQL: Setting the SQLInLine Attribute to FALSE for Out-of-Line Storage

By default, a sub-element is mapped to an embedded object attribute. However, there may be scenarios where out-of-line storage offers better performance. In such cases the `SQLInline` attribute can be set to false, and Oracle XML DB generates an object type with an embedded `REF` attribute. `REF` points to another instance of `XMLType` that corresponds to the XML fragment that gets stored out-of-line. Default `XMLType` tables are also created to store the out-of-line fragments.

Figure 6 illustrates the mapping of a complexType to SQL for out-of-line storage.

Example 5–15 Oracle XML DB XML Schema: complexType Mapping - Setting SQLInLine Attribute to False for Out-of-Line Storage

```
-- Attribute to False for Out-of-Line Storage
-- In this example element Addr's attribute, xdb:SQLInLine, is set to false.
-- The resulting object type OBJ_T2 has a column of type XMLType with an
embedded
-- REF attribute. The REF attribute points to another XMLType instance created
of
-- object type OBJ_T1 in table Addr_tab. Addr_tab has columns Street and City.
The latter
```

```
-- XMLType instance is stored out-of-line.

declare
     doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/emp.xsd"
xmlns:emp="http://www.oracle.com/emp.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb">
     <complexType name = "Employee" xdb:SQLType="OBJ_T2">
        <sequence>
           <element name = "Name" type = "string"/>
           <element name = "Age" type = "decimal"/>
           <element name = "Addr" xdb:SQLInline = "false">
              <complexType xdb:SQLType="OBJ_T1">
                 <sequence>
                    <element name = "Street" type = "string"/>
                    <element name = "City" type = "string"/>
                 </sequence>
              </complexType>
           </element>
        </sequence>
     </complexType>
  </schema>';
begin
        dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

-- On registering this XML schema, Oracle XML DB generates the following types
and XMLType tables:
CREATE TYPE OBJ_T1 AS OBJECT
(
  SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
  Street VARCHAR2(4000),
  City VARCHAR2(4000)
);

CREATE TYPE OBJ_T2 AS OBJECT
(
  SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
  Name VARCHAR2(4000),
  Age NUMBER,
  Addr REF XMLType
);
```

Mapping complexType to SQL: Mapping XML Fragments to Large Objects (LOBs)

You can specify the SQLType for a complex element as a Character Large Object (CLOB) or Binary Large Object (BLOB). Here the entire XML fragment is stored in a LOB attribute. This is useful when parts of the XML document are seldom queried but are mostly retrieved and stored as single pieces. By storing XML fragments as LOBs, you can save on parsing/decomposition/recomposition overheads.

Example 5–16 Oracle XML DB XML Schema: complexType Mapping XML Fragments to LOBs

```
-- In the following example, the XML schema specifies that the XML fragment's
element
-- Addr is using the attribute SQLType="CLOB":
declare
     doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/emp.xsd"
xmlns:emp="http://www.oracle.com/emp.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb">
     <complexType name = "Employee" xdb:SQLType="OBJ_T2">
        <sequence>
           <element name = "Name" type = "string"/>
           <element name = "Age" type = "decimal"/>
           <element name = "Addr" xdb:SQLType = "CLOB">
              <complexType >
                 <sequence>
                    <element name = "Street" type = "string"/>
                    <element name = "City" type = "string"/>
                 </sequence>
              </complexType>
           </element>
        </sequence>
     </complexType>
  </schema>';
begin
       dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

-- On registering this XML schema, Oracle XML DB generates the following types
and XMLType tables:
CREATE TYPE OBJ_T AS OBJECT
(
  SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
  Name VARCHAR2(4000),
  Age NUMBER,
```

```
  Addr CLOB
);
```

Figure 7 Mapping complexType XML Fragments to Character Large Objects (CLOBs)

Oracle XML DB complexType Extensions and Restrictions

In XML schema, `complexTypes` are declared based on `complexContent` and `simpleContent`.

- `simpleContent` is declared as an extension of `simpleType`.
- `complexContent` is declared as one of the following:
  - Base type
  - `complexType` extension
  - `complexType` restriction.

complexType Declarations in XML Schema: Handling Inheritance

For `complexType`, Oracle XML DB handles inheritance in the XML schema as follows:

- ***For complexTypes declared to extend other complexTypes,*** the SQL type corresponding to the base type is specified as the supertype for the current SQL type. Only the additional attributes and elements declared in the sub-complextype are added as attributes to the sub-object-type.

- *For complexTypes declared to restrict other complexTypes,* the SQL type for the sub-complex type is set to be the same as the SQL type for its base type. This is because SQL does not support restriction of object types through the inheritance mechanism. Any constraints are imposed by the restriction in XML schema.

Example 5–17 Inheritance in XML Schema: complexContent as an Extension of complexTypes

```
-- Consider an XML schema that defines a base complexType "Address" and two
extensions
-- "USAddress" and "IntlAddress".
declare
        doc varchar2(3000) := '<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb">
  <xs:complexType name="Address" xdb:SQLType="ADDR_T">
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="USAddress" xdb:SQLType="USADDR_T">
   <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="zip" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="IntlAddress" final="#all" xdb:SQLType="INTLADDR_T">
   <xs:complexContent>
    <xs:extension base="Address">
      <xs:sequence>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
   </xs:complexContent>
  </xs:complexType>
</xs:schema>';
begin
        dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
```

```
end;

-- Note: Type INTLADDR_T is created as a final type because the
-- corresponding complexType specifies the "final" attribute.
-- By default, all complexTypes can be extended and restricted by
-- other types, and hence, all SQL object types are created as not
-- final types.

create type ADDR_T as object (
  SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
  "street" varchar2(4000),
  "city" varchar2(4000)
) not final;

create type USADDR_T under ADDR_T (
 "zip" varchar2(4000)
) not final;

create type INTLADDR_T under ADDR_T (
  "country" varchar2(4000)
) final;
```

Example 5–18 Inheritance in XML Schema: Restrictions in complexTypes

```
-- Consider an XML schema that defines a base complexType Address and a
restricted
-- type LocalAddress that prohibits the specification of country attribute.
declare
        doc varchar2(3000) := '<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:xdb="http://xmlns.oracle.com/xdb">
  <xs:complexType name="Address" xdb:SQLType="ADDR_T">
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="zip" type="xs:string"/>
      <xs:element name="country" type="xs:string" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="LocalAddress" xdb:SQLType="USADDR_T">
    <xs:complexContent>
     <xs:restriction base="Address">
      <xs:sequence>
        <xs:element name="street" type="xs:string"/>
```

```
        <xs:element name="city" type="xs:string"/>
        <xs:element name="zip" type="xs:string"/>
        <xs:element name="country" type="xs:string"
             minOccurs="0" maxOccurs="0"/>
      </xs:sequence>
     </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>';
begin
       dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

-- Since inheritance support in SQL does not support a notion of restriction,
-- the SQL type corresponding to the restricted complexType is a empty subtype
-- of the parent object type.For the above XML schema, the following SQL types
-- are generated:
create type ADDR_T as object (
  SYS_XDBPD$ XDB.XDB$RAW_LIST_T,
  "street" varchar2(4000),
  "city" varchar2(4000),
  "zip" varchar2(4000),
  "country" varchar2(4000)
) not final;

create type USADDR_T under ADDR_T;
```

Mapping complexType: simpleContent to Object Types

A `complexType` based on a `simpleContent` declaration is mapped to an object type with attributes corresponding to the XML attributes and an extra `SYS_XDBBODY` attribute corresponding to the body value. The datatype of the body attribute is based on `simpleType` which defines the body's type.

Example 5–19 XML Schema complexType: Mapping complexType to simpleContent

```
declare
       doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/emp.xsd"
xmlns:emp="http://www.oracle.com/emp.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb">
<complexType name="name" xdb:SQLType="OBJ_T">
  <simpleContent>
    <restriction base = "string">
    </restriction>
```

```
  </simpleContent>
</complexType>
</schema>';
begin
        dbms_xmlschema.registerSchema('http://www.oracle.com/emp.xsd', doc);
end;

-- On registering this XML schema, Oracle XML DB generates the following types
and XMLType tables:
create type OBJ_T as object
(
  SYS_XDBPD$  xdb.xdb$raw_list_t,
  SYS_XDBBODY$ VARCHAR2(4000)
);
```

Mapping complexType: Any and AnyAttributes

Oracle XML DB maps the element declaration, any, and the attribute declaration, anyAttribute, to VARCHAR2 attributes (or optionally to Large Objects (LOBs)) in the created object type. The object attribute stores the text of the XML fragment that matches the any declaration.

- The namespace attribute can be used to restrict the contents so that they belong to a specified namespace.
- The processContents attribute within the any element declaration, indicates the level of validation required for the contents matching the any declaration.

Example 5–20 Oracle XML DB XML Schema: Mapping complexType to Any/AnyAttributes

```
-- This XML schema example declares an any element and maps it to the column
-- SYS_XDBANY$,in object type OBJ_T. This element also declares that the
-- attribute, processContents,skips validating contents that match the any
-- declaration.
declare
doc varchar2(3000) := '<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.oracle.com/any.xsd"
xmlns:emp="http://www.oracle.com/any.xsd"
xmlns:xdb="http://xmlns.oracle.com/xdb">
 <complexType name = "Employee" xdb:SQLType="OBJ_T">
  <sequence>
     <element name = "Name" type = "string" />
     <element name = "Age" type = "decimal"/>
     <any namespace = "http://www/w3.org/2001/xhtml" processContents = "skip"/>
```

```
  </sequence>
 </complexType>
</schema>';
begin
     dbms_xmlschema.registerSchema('http://www.oracle.com/emp.xsd', doc);
end;

-- It results in the following statement:
CREATE TYPE OBJ_T AS OBJECT
(
  SYS_XDBPD$  xdb.xdb$raw_list_t,
  Name VARCHAR2(4000),
  Age NUMBER,
  SYS_XDBANY$ VARCHAR2(4000)
);
```

Handling Cycling Between complexTypes in XML Schema

Cycles in the XML schema are broken while generating the object types, because object types do not allow cycles, by introducing a REF attribute at the point at which the cycle gets completed. Thus part of the data is stored out-of-line yet still belongs to the parent XML document when it is retrieved.

Example 5–21 XML Schema: Cycling Between complexTypes

XML schemas permit cycling between definitions of complexTypes. Figure 5-6 shows this example, where the definition of complexType CT1 can reference another complexType CT2, whereas the definition of CT2 references the first type CT1.

```
-- XML schemas permit cycling between definitions of complexTypes.This is an
example of cycle of length 2:
declare
doc varchar2(3000) := '<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:xdb="http://xmlns.oracle.com/xdb">
  <xs:complexType name="CT1" xdb:SQLType="CT1">
    <xs:sequence>
      <xs:element name="e1" type="xs:string"/>
      <xs:element name="e2" type="CT2"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="CT2" xdb:SQLType="CT2">
    <xs:sequence>
      <xs:element name="e1" type="xs:string"/>
```

```
      <xs:element name="e2" type="CT1"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>';
begin
        dbms_xmlschema.registerSchema('http://www.oracle.com/emp.xsd', doc);
end;
```

SQL types do not allow cycles in type definitions. However, they support weak cycles, that is, cycles involving REF (references) attributes. Therefore, cyclic XML schema definitions are mapped to SQL object types such that any cycles are avoided by forcing `SQLInline="false"` at the appropriate poin. This creates a weak cycle.

```
-- For the preceding XML schema, the following SQL types are generated :
create type CT1 as object
(
  SYS_XDBPD$  xdb.xdb$raw_list_t,
  "e1" varchar2(4000),
  "e2" ref xmltype;
) not final;

create type CT2 as object
(
  SYS_XDBPD$  xdb.xdb$raw_list_t,
  "e1" varchar2(4000),
  "e2" CT1
) not final;
```

Figure 8 Cross Referencing Between Different complexTypes in the Same XML ***Schema***

Example 5–22 XML Schema: Cycling Between complexTypes, Self-Referencing

```
-- Another example of a cyclic complexType involves the declaration of the
-- complexType having a reference to itself. The following is an
-- example of type <SectionT> that references itself:

declare
        doc varchar2(3000) := '<xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:xdb="http://xmlns.oracle.com/xdb">
  <xs:complexType name="SectionT" xdb:SQLType="SECTION_T">
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="body" type="xs:string" xdb:SQLCollType="BODY_COLL"/>
        <xs:element name="section" type="SectionT"/>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:schema>';
begin
        dbms_xmlschema.registerSchema('http://www.oracle.com/section.xsd', doc);
end;

-- The following SQL types are generated.
-- Note: The section attribute is declared as a varray of REFs to XMLType
-- instances. Since there can be more than one occurrence of embedded sections,
-- the attribute is a VARRAY. And it's a VARRAY of REFs to XMLTypes in order to
-- avoid forming a cycle of SQL objects.

create type BODY_COLL as varray(32767) of VARCHAR2(4000);

create type SECTION_T as object
(
  SYS_XDBPD$  xdb.xdb$raw_list_t,
  "title" varchar2(4000),
  "body" BODY_COLL,
  "section" XDB.XDB$REF_LIST_T
) not final;

Further Guidelines for Creating XML Schema-Based XML Tables
Assume that your XML schema, identified by "http://www.oracle.com/PO.xsd", has
been registered. An XMLType table, myPOs, can then be created to store instances
conforming to element, PurchaseOrder, of this XML schema, in an
object-relational format as follows:
CREATE TABLE MyPOs OF XMLTYPE
```

```
   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";
```

Figure 9 illustrates schematically how a `complexTypes` can reference or cycle itself.

Figure 9 complexType Self Referencing Within an XML Schema

Further Guidelines for Creating XML Schema-Based XML Tables

Assume that your XML schema, identified by "`http://www.oracle.com/PO.xsd`", has been registered. An `XMLType` table, `myPOs`, can then be created to store instances conforming to element, `PurchaseOrder`, of this XML schema, in an object-relational format as follows:

```
CREATE TABLE MyPOs OF XMLTYPE
   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOder";
```

Hidden columns are created. These correspond to the object type to which the `PurchaseOrder` element has been mapped. In addition, an `XMLExtra` object column is created to store the top-level instance data such as namespace declarations.

Note: XMLDATA is a pseudo-attribute of XMLType that enables

Specifying Storage Clauses in XMLType CREATE TABLE Statements

To specify storage, the underlying columns can be referenced in the XMLType storage clauses using either Object or XML notation:

- *Object notation:* XMLDATA.<attr1>.<attr2>....

  For example:

```
CREATE TABLE  MyPOs OF XMLTYPE
   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder"
         lob (xmldata.lobattr) STORE AS (tablespace ...);
```

- *XML notation*: extractValue(xmltypecol, '/attr1/attr2')

  For example:

```
CREATE TABLE MyPOs OF XMLTYPE
   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder"
         lob (ExtractValue(MyPOs, '/lobattr')) STORE AS (tablespace ...);
```

Referencing XMLType Columns Using CREATE INDEX

As shown in the preceding examples, columns underlying an XMLType column can be referenced using either an *object* or *XML* notation in the CREATE TABLE statements. The same is true in CREATE INDEX statements:

```
CREATE INDEX ponum_idx ON MyPOs (xmldata.ponum);
CREATE INDEX ponum_idx ON MyPOs p (ExtractValue(p, '/ponum');
```

Specifying Constraints on XMLType Columns

Constraints can also be specified for underlying XMLType columns, using either the *object* or *XML* notation:

- *Object notation*

```
CREATE TABLE MyPOs OF XMLTYPE
ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder"
(unique(xmldata.ponum));
```

- *XML notation*

```
CREATE TABLE MyPOs P OF XMLTYPE
ELEMENT
"http://www.oracle.com/PO.xsd#PurchaseOrder"(unique(ExtractValue(p,'/ponum')
);
```

Inserting New Instances into XMLType Columns

New instances can be inserted into an XMLType columns as follows:

```
INSERT INTO MyPOs VALUES
     (xmltype.createxml('<PurchaseOrder>.....</PurchaseOrder>'));
```

Query Rewrite with XML Schema-Based Object-Relational Storage

What is Query Rewrite?

When the XMLType is stored in structured storage (object-relationally) using an XML schema and queries using XPath are used, they are rewritten to go directly to the underlying object-relational columns. This enables the use of B*Tree or other indexes, if present on the column, to be used in query evaluation by the Optimizer. This query rewrite mechanism is used for XPath's in SQL functions such as `existsNode()`, `extract()`, `extractValue()`, and `updateXML()`. This enables the XPath to be evaluated against the XML document without having to ever construct the XML document in memory.

Example 5–23 Query Rewrite

For example a query such as:

```
SELECT VALUE(p) FROM MyPOs p
     WHERE extractValue(value(p),'/PurchaseOrder/Company') = 'Oracle';
```

is trying to get the value of the `Company` element and compare it with the literal 'Oracle'. Since the `MyPOs` table has been created with XML schema-based object-relational storage, the `extractValue` operator gets rewritten to the underlying relational column which stores the company information for the `purchaseorder`.

Thus the preceding query is rewritten to the following:

```
SELECT VALUE(p) FROM MyPOs p
```

```
WHERE p.xmldata.company = 'Oracle';
```

If there was a regular index created on the Company column, such as:

```
CREATE INDEX company_index ON MyPos e
      (extractvalue(value(e),'/PurchaseOrder/Company'));
```

then the preceding query would use the index for its evaluation.

When Does Query Rewrite Occur?

Query rewrite happens for the following SQL functions,

- `extract()`
- `existsNode()`
- `extractValue`
- `updateXML`

The rewrite happens for these SQL functions which may be present in any expression in a query, DML, or DDL statements. For example, you can use the `extractValue()` to create indexes on the underlying relational columns.

Example 5–24 SELECT Statement and Query Rewrites

```
-- This example gets the existing purchase orders

SELECT EXTRACTVALUE(value(x),'/PurchaseOrder/Company')
  FROM MYPOs x
  WHERE EXISTSNODE(value(x),'/PurchaseOrder/Item[1]/Part') = 1;
```

Here are some examples of statements that get rewritten to use the underlying columns:

Example 5–25 DML Statement and Query Rewrites

```
-- This example deletes all purchaseorders where the company is not Oracle:

DELETE FROM MYPOs x
 WHERE EXTRACTVALUE(value(x),'/PurchaseOrder/Company') = 'Oracle Corp';
```

Example 5–26 CREATE INDEX Statement and Query Rewrites

```
-- This example creates an index on the Company column - since this is stored
-- object relationally and the query rewrite happens, a regular index on the
-- underlying relational column will be created:

CREATE INDEX company_index ON MyPos e
      (extractvalue(value(e),'/PurchaseOrder/Company'));
```

In this case, if the rewrite of the SQL functions results in a simple relational column, then the index is turned into a BTree or a domain index on the column, rather than a function-based index.

What XPath Expressions are Rewritten?

XPath involving simple expressions with no wild cards or descendant axes get rewritten. The XPath may select an element or an attribute node. Predicates are supported and get rewritten into SQL predicates.

Table 5–10 lists the kinds of XPath expressions that can be translated into underlying SQL queries in this release.

Table 5–10 Supported XPath Expressions For Translation to Underlying SQL Queries

XPath Expression for Translation	Description
Simple XPath expressions: /PurchaseOrder/@PurchaseDate /PurchaseOrder/Company	Involves traversals over object type attributes only, where the attributes are simple scalar or object types themselves. The only axes supported are the child and the attribute axes.
Collection traversal expressions: /PurchaseOrder/Item/Part	Involves traversal of collection expressions. The only axes supported are child and attribute axes. Collection traversal is not supported if the SQL operator is used during `CREATE INDEX` or `updateXML()`.
Predicates: [Company="Oracle"]	Predicates in the XPath are rewritten into SQL predicates. Predicates are not rewritten for `updateXML()`
List indexe: lineitem[1]	Indexes are rewritten to access the n'th item in a collection. These are not rewritten for `updateXML()`.

Unsupported XPath Constructs The following XPath constructs do not get rewritten:

- XPath Functions
- XPath Variable references

- All axis other than child and attribute axis
- Wild card and descendant expressions
- UNION operations

Unsupported XMLSchema Constructs The following XML Schema constructs are not supported. This means that if the XPath expression includes nodes with the following XML Schema construct then the entire expression will not get rewritten:

- XPath expressions accessing children of elements containing open content, namely `any` content. When nodes contain `any` content, then the expression cannot be rewritten, except when the `any` targets a namespace other than the namespace specified in the XPath. `any` attributes are handled in a similar way.
- CLOB storage. If the XML schema maps part of the element definitions to an SQL CLOB, then XPath expressions traversing such elements are not supported.
- Enumeration types.
- Substitutable elements.

Non-default mapping of scalar types. For example, number types mapped to native storage, such as native integers, and so on.

- Child access for inherited `complexTypes` where the child is not a member of the declared `complexType`.

  For example, consider the case where we have a `address complexType` which has a `street` element. We can have a derived type called `shipAddr` which contains `shipmentNumber` element. If the `PurchaseOrder` had an address element of type address, then an XPath like `"/PurchaseOrder/address/street"` would get rewritten whereas `"/PurchaseOrder/address/shipmentNumber"` would not.

- Non-coercible datatype operations, such as a boolean added with a number.

How are the XPaths Rewritten?

The following sections use the same `purchaseorder` schema explained earlier in the chapter to explain how functions get rewritten.

Example 5–27 Rewrting XPaths During Object Type Generation

```
-- Consider the following purchaseorder schema:

declare
```

```
        doc varchar2(1000) := '<schema
targetNamespace="http://www.oracle.com/PO.xsd"
xmlns:po="http://www.oracle.com/PO.xsd" xmlns="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
 <complexType name="PurchaseOrderType">
  <sequence>
   <element name="PONum" type="decimal"/>
   <element name="Company">
    <simpleType>
     <restriction base="string">
      <maxLength value="100"/>
     </restriction>
    </simpleType>
   </element>
   <element name="Item" maxOccurs="1000">
    <complexType>
     <sequence>
      <element name="Part">
       <simpleType>
        <restriction base="string">
         <maxLength value="1000"/>
        </restriction>
       </simpleType>
      </element>
      <element name="Price" type="float"/>
     </sequence>
    </complexType>
   </element>
  </sequence>
 </complexType>
 <element name="PurchaseOrder" type="po:PurchaseOrderType"/>
</schema>';
begin
        dbms_xmlschema.registerSchema('http://www.oracle.com/PO.xsd', doc);
end;

-- A table is created conforming to this schema
CREATE TABLE MyPOs OF XMLTYPE

   ELEMENT "http://www.oracle.com/PO.xsd#PurchaseOrder";

-- The inserted XML document is partially validated against the schema before
-- it is inserted.
insert into MyPos values (xmltype('<PurchaseOrder
xmlns="http://www.oracle.com/PO.xsd"
```

```
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.oracle.com/PO.xsd
http://www.oracle.com/PO.xsd">
     <PONum>1001</PONum>
     <Company>Oracle Corp</Company>
     <Item>
       <Part>9i Doc Set</Part>
       <Price>2550</Price>
     </Item>
     <Item>
       <Part>8i Doc Set</Part>
       <Price>350</Price>
     </Item>
</PurchaseOrder>'));
```

Since the XML schema did not specify anything about maintaining the ordering, the default is to maintain the ordering and DOM fidelity. Hence the types have SYS_XDBPD$ attribute to store the extra information needed to maintain the ordering of nodes and to capture extra items such as comments, processing instructions and so on.

The SYS_XDBPD$ attribute also maintains the existential information for the elements (that is, whether the element was present or not in the input document). This is needed for elements with scalar content, since they map to simple relational columns. In this case, both empty and missing scalar elements map to NULL values in the column and only the SYS_XDBPD$ attribute can help distinguish the two cases. The query rewrite mechanism takes into account the presence or absence of the SYS_XDBPD$ attribute and rewrites queries appropriately.

Assuming that this XML schema is registered with the schema URL:

http://www.oracle.com/PO.xsd

you can create the po_tab table with that schema:

```
CREATE TABLE po_tab OF XMLTYPE
    XMLSCHEMA "http://www.oracle.com/PO.xsd" ELEMENT "PurchaseOrder";
```

Now, this table has a hidden XMLData column that is of type "PurchaseOrder_T" which stores the actual data.

Rewriting XPath Expressions: Mapping Types and Issues

XPath expression mapping types and topics are described in the following sections:

- "Mapping for Simple XPath"

- "Mapping for Scalar Nodes"
- "Mapping of Predicates"
- "Mapping of Collection Predicates"
- "Document Ordering with Collection Traversals"
- "Collection Index"
- "Non-Satisfiable XPath Expressions"
- "Namespace Handling"
- "Date Format Conversions"

Mapping for Simple XPath A rewrite for a simple XPath involves accessing the attribute corresponding to the XPath expression. Table 5–11 lists the XPath map:

Table 5–11 Simple XPath Mapping for purchaseOrder XML Schema

XPath Expression	Maps to
/PurchaseOrder	column XMLData
/PurchaseOrder/@PurchaseDate	column XMLData."PurchaseDate"
/PurchaseOrder/PONum	column XMLData."PONum"
/PurchaseOrder/Item	elements of the collection XMLData."Item"
/PurchaseOrder/Item/Part	attirbute "Part" in the collection XMLData."Item"

Mapping for Scalar Nodes An XPath expression can contain a `text()` operator which maps to the scalar content in the XML document. When rewriting, this maps directly to the underlying relational columns.

For example, the XPath expression "`/PurchaseOrder/PONum/text()`" maps to the SQL column `XMLData."PONum"` directly.

A NULL value in the `PONum` column implies that the text value is not available, either because the text node was not present in the input document or the element itself was missing. This is more efficient than accessing the scalar element, since we do not need to check for the existance of the element in the `SYS_XBDPD$` attribute.

For example, the XPath "`/PurchaseOrder/PONum`" also maps to the SQL attribute `XMLData."PONum"`,

However,in this case, query reqrite also has to check for the existance of the element itself, using the SYS_XDBPD$ in the XMLData column.

Mapping of Predicates Predicates are mapped to SQL predicate expressions.

Example 5–28 Maping Predicates

```
-- For example the predicate in the XPath expression:

/PurchaseOrder[PONum=1001 and Company = "Oracle Corp"]
-- maps to the SQL predicate:
( XMLData."PONum" = 20 and XMLData."Company" = "Oracle Corp")

-- For example, the following query will get rewritten to the object-relational
-- equivalent, and will not require Functional evaluation of the XPath.
select extract(value(p),'/PurchaseOrder/Item').getClobval()
   from mypos p
   where existsNode(value(p),'/PurchaseOrder[PONum=1001 and Company = "Oracle
Corp"]') =1;
```

Mapping of Collection Predicates XPath expressions may involve relational operators with collection expressions. In Xpath 1.0, conditions involving collections are existential checks. In other words, even if one member of the collection satisfies the condition, the expression is true.

Example 5–29 Mapping Collection Predicates

```
-- For example the collection predicate in the XPath:
/PurchaseOrder[Items/Price > 200]
-- maps to a SQL collection expression:
EXISTS ( SELECT null
    FROM   TABLE (XMLDATA."Item") x
    WHERE  x."Price" > 200 )

-- For example, the following query will get rewritten to the object-relational
-- equivalent.
select extract(value(p),'/PurchaseOrder/Item').getClobval()
   from mypos p
   where existsNode(value(p),'/PurchaseOrder[Item/Price > 400]') = 1;
```

More complicated rewrites occur when you have a collection <condition> collection. In this case, if at least one combination of nodes from these two collection arguments satisfy the condition, then the predicate is deemed to be satisfied.

Example 5–30 Mapping Collection Predicates, Using existsNode()

```
-- For example, consider a fictitious XPath which checks to see if a
-- Purchaseorder has Items such that the price of an item is the same as
-- some part number:

/PurchaseOrder[Items/Price = Items/Part]
-- maps to a SQL collection expression:
   EXISTS ( SELECT null
            FROM   TABLE (XMLDATA."Item") x
            WHERE  EXISTS ( SELECT null
                            FROM  TABLE(XMLDATA."Item") y
                            WHERE  y."Part" = x."Price"))

-- For example, the following query will get rewritten to the object-relational
-- equivalent.
select extract(value(p),'/PurchaseOrder/Item').getClobval()
    from mypos p
    where existsNode(value(p),'/PurchaseOrder[Item/Price = Item/Part]') = 1;
```

Document Ordering with Collection Traversals Most of the rewrite preserves the original document ordering. However, since the SQL system does not guarantee ordering on the results of subqueries, when selecting elements from a collection using the `extract()` function, the resultant nodes may not be in document order.

Example 5–31 Document Ordering with Collection Traversals

```
-- For example:
SELECT extract(value(p),'/PurchaseOrder/Item[Price>2100]/Part')
FROM mypos p;
-- gets rewritten to use subqueries as shown below:
SELECT (SELECT XMLAgg( XMLForest(x."Part" AS "Part"))
        FROM   TABLE (XMLData."Item") x
        WHERE  x."Price" > 2100 )
        FROM po_tab p;
```

Though in most cases, the result of the aggregation would be in the same order as the collection elements, this is not guaranteed and hence the results may not be in document order. This is a limitation that may be fixed in future releases.

Collection Index An XPath expression can also access a particular index of a collection For example, `"/PurchaseOrder/Item[1]/Part"` is rewritten to extract out the first Item of the collection and then access the Part attribute within that.

If the collection has been stored as a VARRAY, then this operation retrieves the nodes in the same order as present in the original document. If the mapping of the collection is to a nested table, then the order is undetermined. If the VARRAY is stored as an Ordered Collection Table (OCT) , (the default for the tables created by the schema compiler, if `storeVarrayAsTable="true"` is set), then this collection index access is optimized to use the IOT index present on the VARRAY.

Non-Satisfiable XPath Expressions An XPath expression can contain references to nodes that cannot be present in the input document. Such parts of the expression map to SQL NULLs during rewrite. For example the XPath expression: `"/PurchaseOrder/ShipAddress"` cannot be satisified by any instance document conforming to the `PO.xsd` XML schema, since the XML schema does not allow for `ShipAddress` elements under `PurchaseOrder`. Hence this expression would map to a SQL NULL literal.

Namespace Handling Namespaces are handled in the same way as the function-based evaluation. For schema based documents, if the function (like EXISTSNODE/EXTRACT) does not specify any namespace parameter, then the target namespace of the schema is used as the default namespace for the XPath expression.

Example 5–32 Handling Namespaces

```
-- For example, the XPath expression /PurchaseOrder/PONum is treated as
-- /a:PurchaseOrder/a:PONum with xmlns:a="http://www.oracle.com/PO.xsd" if
-- the SQL function does not explicitly specify the namespace prefix and
mapping. In other words:
SELECT * FROM po_tab p
   WHERE  EXISTSNODE(value(p), '/PurchaseOrder/PONum') = 1;

-- is equivalent to the query:
SELECT * FROM po_tab p
   WHERE  EXISTSNODE(value(p),'/PurchaseOrder/PONum',
       'xmlns="http://www.oracle.com/PO.xsd') = 1;
```

When performing query rewrite, the namespace for a particular element is matched with that of the XML schema definition. If the XML schema contains `elementFormDefault="qualified"` then each node in the XPath expression must target a namespace (this can be done using a default namespace specification or by prefixing each node with a namespace prefix).

If the `elementFormDefault` is unqualified (which is the default), then only the node that defines the namespace should contain a prefix. For instance if the `PO.xsd`

had the element form to be unqualified, then the `existsNode()` function should be rewritten as:

```
EXISTSNODE(value(p),'/a:PurchaseOrder/PONum',
                    'xmlns:a="http://www.oracle.com/PO.xsd") = 1;
```

> **Note:** For the case where `elementFormDefault` is unqualified, omitting the namespace parameter in the SQL function `existsNode()` in hte preceding example, would cause each node to default to the target namespa. This would not match the XMLSchema definition and consequently would not return any result. This is true whether the function is rewritten or not.

Date Format Conversions The default date formats are different for XML schema and SQL. Consequently, when rewriting XPath expressions involving comparsions with dates, you need to use XML formats.

Example 5–33 Date Format Conversions

```
-- For example, the expression:
[@PurchaseDate="2002-02-01"]

-- cannot be simply rewritten as:
XMLData."PurchaseDate" = "2002-02-01"

-- since the default date format for SQL is not YYYY-MM-DD. Hence during
-- rewrite,the XML format string is added to convert text values into date
-- datatypes correctly.
-- Thus the preceding predicate would be rewritten as:

XMLData."PurchaseDate" = TO_DATE("2002-02-01","SYYYY-MM-DD");
```

Similarly when converting these columns to text values (needed for `extract()`, and so on), XML format strings are added to convert them to the same date format as XML.

XPath Expression Rewrites for existsNode()

`existsNode()` returns a numerical value 0 or 1 indicating if the XPath returns any nodes (`text()` or element nodes). Based on the mapping discussed in the earlier section, an `existsNode` simply checks if a scalar element is non-null in the case

where the XPath targets a `text()` node or a `non-scalar` node and checks for the existance of the element using the `SYS_XDBPD$` otherwise. If the `SYS_XDBPD$` attribute is absent, then the existance of a `scalar` node is determined by the null information for the scalar column.

existsNode Mapping with Document Order Maintained Table 5–12 shows the mapping of various XPaths in the case of `existsNode()` when document ordering is preserved, that is, when `SYS_XDBPD$` exists and `maintainDOM="true"` in the schema document.

Table 5–12 XPath Mapping for existsNode() with Document Ordering Preserved

XPath Expression	Maps to
/PurchaseOrder	CASE WHEN XMLData IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/@PurchaseDate	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, 'PurchaseDate') = 1 THEN 1 ELSE 0 END
/PurchaseOrder/PONum	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, 'PONum') = 1 THEN 1 ELSE 0 END
/PurchaseOrder[PONum = 2100]	CASE WHEN XMLData."PONum" = 2100 THEN 1 ELSE 0
/PurchaseOrder[PONum = 2100]/@PurchaseDate	CASE WHEN XMLData."PONum" = 2100 AND Check_Node_Exists(XMLData.SYS_XDBPD$, 'PurchaseDate') = 1 THEN 1 ELSE 0 END
/PurchaseOrder/PONum/text()	CASE WHEN XMLData."PONum" IS NOT NULL THEN 1 ELSE 0
/PurchaseOrder/Item	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE value(x) IS NOT NULL) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE Check_Node_Exists(x.SYS_XDBPD$, 'Part') = 1) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part/text()	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE x."Part" IS NOT NULL) THEN 1 ELSE 0 END

Example 5–34 existsNode Mapping with Document Order Maintained

```
-- Using the preceding mapping, a query which checks whether the purchaseorder

-- with number 2100 contains a part with price greater than 2000:
SELECT count(*)
FROM   mypos p
WHERE EXISTSNODE(value(p),'/PurchaseOrder[PONum=1001 and Item/Price > 2000]')=
1;

-- would become:
SELECT count(*)
FROM   mypos p
WHERE  CASE WHEN
           p.XMLData."PONum" = 1001 AND
           EXISTS ( SELECT NULL
                    FROM   TABLE ( XMLData."Item") p
                    WHERE  p."Price" > 2000 )) THEN 1 ELSE 0 END  = 1;

-- The CASE expression gets further optimized due to the constant relational
-- equality expressions and this query becomes:
SELECT count(*)
FROM   mypos p
WHERE  p.XMLData."PONum" = 1001 AND
       EXISTS ( SELECT NULL
                FROM   TABLE ( p.XMLData."Item") x
                WHERE  x."Price" > 2000 );
-- which would use relational indexes for its evaluation, if present on the Part
and PONum columns.
```

existsNode Mapping Without Maintaining Document Order If the `SYS_XDBPD$` does not exist (that is, if the XML schema specifies `maintainDOM="false"`) then NULL scalar columns map to non-existant scalar elements. Hence you do not need to check for the node existance using the `SYS_XDBPD$` attribute. Table 5–13 shows the mapping of `existsNode()` in the absence of the SYS_XDBPD$ attribute.

Table 5–13 XPath Mapping for existsNode Without Document Ordering

XPath Expression	Maps to
/PurchaseOrder	CASE WHEN XMLData IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/@PurchaseDate	CASE WHEN XMLData.'PurchaseDate' IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/PONum	CASE WHEN XMLData."PONum" IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder[PONum = 2100]	CASE WHEN XMLData."PONum" = 2100 THEN 1 ELSE 0 END

Table 5–13 XPath Mapping for existsNode Without Document Ordering (Cont.)

XPath Expression	Maps to
/PurchaseOrder[PONum = 2100]/@PurchaseOrderDate	CASE WHEN XMLData."PONum" = 2100 AND XMLData."PurchaseDate" NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/PONum/text()	CASE WHEN XMLData."PONum" IS NOT NULL THEN 1 ELSE 0 END
/PurchaseOrder/Item	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE value(x) IS NOT NULL) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE x."Part" IS NOT NULL) THEN 1 ELSE 0 END
/PurchaseOrder/Item/Part/text()	CASE WHEN EXISTS (SELECT NULL FROM TABLE (XMLData."Item") x WHERE x."Part" IS NOT NULL) THEN 1 ELSE 0 END

Rewrite for extractValue()

`extractValue()` is a shortcut for extracting `text` nodes and attributes using `extract()` and then using a `getStringVal()` or `getNumberVal()` to get the scalar content. `extractValue` returns the `text` nodes for scalar elements or the values of `attribute` nodes. `extractValue()` cannot handle returning multiple values or non-scalar elements.

Table 5–14 shows the mapping of various XPath expressions in the case of extractValue(). If an XPath expression targets an element, `extractValue` retrieves the `text` node child of the element. Thus the two XPath expressions, `/PurchaseOrder/PONum` and `/PurchaseOrder/PONum/text()` are handled identically by `extractValue` and both of them retrieve the scalar content of `PONum`.

Table 5–14 XPath Mapping for extractValue()

XPath Expression	Maps to
/PurchaseOrder	Not supported - ExtractValue can only retrieve values for scalar elements and attributes
/PurchaseOrder/@PurchaseDate	XMLData."PurchaseDate"
/PurchaseOrder/PONum	XMLData."PONum"

Table 5–14 XPath Mapping for extractValue() (Cont.)

XPath Expression	Maps to
/PurchaseOrder[PONum = 2100]	(SELECT TO_XML(x.XMLData) FROM Dual WHERE x."PONum" = 2100)
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT x.XMLData."PurchaseDate") FROM Dual WHERE x."PONum" = 2100)
/PurchaseOrder/PONum/text()	XMLData."PONum"
/PurchaseOrder/Item	Not supported - ExtractValue can only retrieve values for scalar elements and attributes
/PurchaseOrder/Item/Part	Not supported - ExtractValue cannot retireve multiple scalar values
/PurchaseOrder/Item/Part/text()	Not supported - ExtractValue cannot retireve multiple scalar values

Example 5–35 Rewriting extractValue()

```
-- For example, an SQL query such as:
SELECT ExtractValue(value(p),'/PurchaseOrder/PONum')
   FROM   mypos p
   WHERE  ExtractValue(value(p),'/PurchaseOrder/PONum') = 1001;

-- would become:
SELECT p.XMLData."PONum"
   FROM   mypos p
   WHERE  p.XMLData."PONum" = 1001;
```

Since it gets rewritten to simple scalar columns, indexes if any, on the `PONum` attribute may be used to satisfy the query.

Creating Indexes `ExtractValue` can be used in index expressions. If the expression gets rewritten into scalar columns, then the index is turned into a BTree index instead of a function-based index.

Example 5–36 Creating Indexes

```
-- For example:
create index my_po_index on mypos x
  (Extract(value(x),'/PurchaseOrder/PONum/text()').getnumberval());

-- would get rewritten into:
create index my_po_index on mypos x ( x.XMLData."PONum");
```

```
-- and thus becomes a regular BTree index. This is useful, since
-- unlike a functional index, the same index can now satisfy queries which
target the column such as:
EXISTSNODE(value(x),'/PurchaseOrder[PONum=1001]') = 1;

-- and thus becomes a regular B*Tree index. This is useful, since
-- unlike a functional index, the same index can now satisfy queries which
target the column such as:
EXISTSNODE(value(x),'/PurchaseOrder[PONum=1001]') = 1;
```

Rewrite for extract()

`extract()` retrieves the results of XPath as XML. The rewrite for `extract()` is similar to that of `extractValue()` for those Xpath expressions involving `text` nodes.

Extract Mapping with Document Order Maintained Table 5–15 shows the mapping of various XPath in the case of `extract()` when document order is preserved (that is, when `SYS_XDBPD$` exists and `maintainDOM="true"` in the schema document).

> **Note:** The examples show `XMLElement` and `XMLForest` with an empty alias string "" to indicate that you create a XML instance with only `text` values. This is shown for illustration only.

Table 5–15 XPath Mapping for extract() with Document Ordering Preserved

XPath	Maps to
/PurchaseOrder	XMLForest(XMLData as "PurchaseOrder")
/PurchaseOrder/@PurchaseDate	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, 'PurchaseDate') = 1 THEN XMLElement("", XMLData."PurchaseDate") ELSE NULL END
/PurchaseOrder/PONum	CASE WHEN Check_Node_Exists(XMLData.SYS_XDBPD$, 'PONum') = 1 THEN XMLElement("PONum", XMLData."PONum") ELSE NULL END
/PurchaseOrder[PONum = 2100]	(SELECT XMLForest(XMLData as "PurchaseOrder") FROM Dual WHERE x."PONum" = 2100)

Table 5–15 XPath Mapping for extract() with Document Ordering Preserved (Cont.)

XPath	Maps to
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT CASE WHEN Check_Node_Exists(x.XMLData.SYS_XDBPD$,'PurchaseDate") = 1 THEN XMLElement("", XMLData."PurchaseDate") ELSE NULL END FROM Dual WHERE x."PONum" = 2100)
/PurchaseOrder/PONum/text()	XMLElement("", XMLData.PONum)
/PurchaseOrder/Item	(SELECT XMLAgg(XMLForest(value(p) as "Item")) FROM TABLE (x.XMLData."Item") p WHERE value(p) IS NOT NULL)
/PurchaseOrder/Item/Part	(SELECT XMLAgg(CASE WHEN Check_Node_Exists(p.SYS_XDBPD$,'Part") = 1 THEN XMLForest(p."Part" as "Part") ELSE NULL END) FROM TABLE (x.XMLData."Item") p)
/PurchaseOrder/Item/Part/text()	(SELECT XMLAgg(XMLElement(" ", p."Part")) FROM TABLE (x.XMLData."Item") x)

Example 5–37 XPath Mapping for extract() with Document Ordering Preserved

```
-- Using the mapping in Table 5-15, a query that extracts the PONum element
-- where the purchaseorder contains a part with price greater than 2000:
SELECT Extract(value(p),'/PurchaseOrder[Item/Part > 2000]/PONum')
FROM   po_tab p;

-- would become:
SELECT (SELECT CASE WHEN Check_Node_Exists(p.XMLData.SYS_XDBPD$, 'PONum')  = 1
                    THEN XMLElement("PONum", p.XMLData."PONum")
                    ELSE NULL END)
        FROM  DUAL
        WHERE  EXISTS( SELECT NULL
                       FROM   TABLE ( XMLData."Item") p
                       WHERE  p."Part" > 2000)
       )
FROM po_tab p;

-- Check_Node_Exists is an internal function that is for illustration purposes
```

```
-- only.
```

Extract Mapping Without Maintaining Document Order If the SYS_XDBPD$ does not exist, that is, if the XML schema specifies `maintainDOM="false"`, then NULL scalar columns map to non-existant scalar elements. Hence you do not need to check for the node existance using the SYS_XDBPD$ attribute. Table 5–16 shows the mapping of `existsNode()` in the absence of the SYS_XDBPD$ attribute.

Table 5–16 XPath Mapping for extract() Without Document Ordering Preserved

XPath	Equivalent to
/PurchaseOrder	XMLForest(XMLData AS "PurchaseOrder")
/PurchaseOrder/@PurchaseDate	XMLForest(XMLData."PurchaseDate" AS "")
/PurchaseOrder/PONum	XMLForest(XMLData."PONum" AS "PONum")
/PurchaseOrder[PONum = 2100]	(SELECT XMLForest(XMLData AS "PurchaseOrder") FROM Dual WHERE x."PONum" = 2100)
/PurchaseOrder[PONum = 2100]/@PurchaseDate	(SELECT XMLForest(XMLData."PurchaseDate" AS "") FROM Dual WHERE x."PONum" = 2100)
/PurchaseOrder/PONum/text()	XMLForest(XMLData.PONum AS "")
/PurchaseOrder/Item	(SELECT XMLAgg(XMLForest(value(p) as "Item") FROM TABLE (x.XMLData."Item") p WHERE value(p) IS NOT NULL)
/PurchaseOrder/Item/Part	(SELECT XMLAgg(XMLForest(p."Part" AS "Part") FROM TABLE (x.XMLData."Item") p)
/PurchaseOrder/Item/Part/text()	(SELECT XMLAgg(XMLForest(p. "Part" AS "Part")) FROM TABLE (x.XMLData."Item") p)

Optimizing Updates Using updateXML()

A regular update using `updateXML()` involves updating a value of the XML document and then replacing the whole document with the newly updated document.

When `XMLType` is stored object relationally, using XML schema mapping, updates are optimized to directly update pieces of the document. For example, updating the

PONum element value can be rewritten to directly update the XMLData.PONum column instead of materializing the whole document in memory and then performing the update.

`updateXML()` must satisfy the following conditions for it to use the optimization:

- The `XMLType` column supplied to `updateXML()` must be the same column being updated in the SET clause. For example:

```
UPDATE po_tab p SET value(p) = updatexml(value(p),...);
```

- The `XMLType` column must have been stored object relationally using Oracle XML DB's XML schema mapping.
- The XPath expressions must not involve any predicates or collection traversals.
- There must be no duplicate scalar expressions.
- All XPath arguments in the `updateXML()` function must target only scalar content, that is, text nodes or attributes - For example:

```
UPDATE po_tab p SET value(p) =
    updatexml(value(p),'/PurchaseOrder/@PurchaseDate','2002-01-02',
                       '/PurchaseOrder/PONum/text()', 2200);
```

If all the preceding conditions are satisfied, then the updateXML is rewritten into a simple relational update. For example,

```
UPDATE po_tab p SET value(p) =
       updatexml(value(p),'/PurchaseOrder/@PurchaseDate','2002-01-02',
                          '/PurchaseOrder/PONum/text()', 2200);
```

becomes,

```
UPDATE po_tab p
  SET p.XMLData."PurchaseDate" = TO_DATE('2002-01-02','SYYYY-MM-DD'),
      p.XMLData."PONum" = 2100;
```

DATE Conversions Date datatypes such as Date, gMonth, gDate etc.., have different format in the XMLSchema and SQL system. In such cases, if the updateXML has a string value for these columns, the rewrite automatically puts the XML format string to convert the string value correctly. Thus string value specified for date columns, must match the XML date format and not the SQL date format.

Creating Default Tables During XML Schema Registration

As part of XML schema registration, you can also create default tables. Default tables are most useful when XML instance documents conforming to this XML schema are inserted through APIs that do not have any table specification, such as with FTP or HTTP. In such cases, the XML instance is inserted into the default table.

If you have given a value for attribute `defaultTable`, the `XMLType` table is created with that name. Otherwise it gets created with an internally generated name.

Further, any text specified using the `tableProps` and `columnProps` attribute are appended to the generated CREATE TABLE statement.

Ordered Collections in Tables (OCTs)

Arrays in XML schemas (elements with `maxOccurs > 1`) are usually stored in VARRAYs, which can be stored either in a Large Object (LOB) or in a separate store table, similar to a nested table.

Note: When elements of a VARRAY are stored in a separate table, the VARRAY is referred to as an Ordered Collection in Tables (OCT). In the following paragraphs, references to OCT also assume that you are using Index Organized Table (IOT) storage for the "store" table.

This allows the elements of a VARRAY to reside in a separate table based on an IOT. The primary key of the table is (NESTED_TABLE_ID, ARRAY_INDEX). NESTED_TABLE_ID is used to link the element with their containing parents while the ARRAY_INDEX column keeps track of the position of the element within the collection.

Using OCT for VARRAY Storage

There are two ways to specify an OCT storage:

- By means of the schema attribute `"storeVarrayAsTable"`. By default this is "false" and VARRAYs are stored in a LOB. If this is set to `"true"`, all VARRAYs, all elements that have `maxOccurs > 1`, will be stored as OCTs.

- By explicitly specifying the storage using the "tableProps" attribute. The exact SQL needed to create an OCT can be used as part of the tableProps attribute:

  "VARRAY xmldata.<array> STORE AS TABLE <myTable> ((PRIMARY KEY (NESTED_ TABLE_ID, ARRAY_INDEX)) ORGANIZATION INDEX)"

The advantages of using OCTs for VARRAY storage include faster access to elements and better queriability. Indexes can be created on attributes of the element and these can aid in better execution for query rewrite.

Cyclical References Between XML Schemas

XML schema documents can have cyclic dependencies that can prevent them from being registered one after the other in the usual manner. Examples of such XML schemas follow:

Example 5–38 Cyclic Dependencies

```
-- A schema thats including another schema cannot be created
-- if the included schema does not exist.
begin dbms_xmlschema.registerSchema('xm40.xsd',
'<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:my="xm40"
targetNamespace="xm40">
  <include schemaLocation="xm40a.xsd"/>
  <!-- Define a global complextype here -->
  <complexType name="Company">
    <sequence>
      <element name="Name" type="string"/>
      <element name="Address" type="string"/>
    </sequence>
  </complexType>
  <!-- Define a global element depending on included schema -->
  <element name="Emp" type="my:Employee"/>
</schema>',
true, true, false, true); end;
/

-- It can however be created with the FORCE option
begin dbms_xmlschema.registerSchema('xm40.xsd',

'<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:my="xm40"
targetNamespace="xm40">
  <include schemaLocation="xm40a.xsd"/>
```

```
  <!-- Define a global complextype here -->
  <complexType name="Company">
    <sequence>
      <element name="Name" type="string"/>
      <element name="Address" type="string"/>
    </sequence>
  </complexType>
  <!-- Define a global element depending on included schema -->
  <element name="Emp" type="my:Employee"/>
</schema>',
true, true, false, true, true); end;
/

-- Attempt to use this schema will try recompiling
-- and fail.
create table foo of sys.xmltype xmlschema "xm40.xsd" element "Emp";

-- Now create the 2nd schema with FORCE option
-- This should also make the 1st schema VALID.

begin dbms_xmlschema.registerSchema('xm40a.xsd',
'<schema xmlns="http://www.w3.org/2001/XMLSchema" xmlns:my="xm40"
targetNamespace="xm40">
  <include schemaLocation="xm40.xsd"/>
  <!-- Define a global complextype here -->
  <complexType name="Employee">
    <sequence>
      <element name="Name" type="string"/>
      <element name="Age" type="positiveInteger"/>
      <element name="Phone" type="string"/>
    </sequence>
  </complexType>
  <!-- Define a global element depending on included schema -->
  <element name="Comp" type="my:Company"/>
</schema>',
true, true, false, true, true); end;
/
-- Both can be used to create tables etc.
create table foo of sys.xmltype xmlschema "xm40.xsd" element "Emp";
create table foo2 of sys.xmltype xmlschema "xm40a.xsd" element "Comp";
```

To register both these XML schemas which have a cyclic dependency on each other, you must use the FORCE parameter in `DBMS_XMLSCHEMA.registerSchema` as follows:

1. Step 1 : Register "s1.xsd" in FORCE mode:

```
dbms_xmlschema.registerSchema("s1.xsd", "<schema ...", ..., force => true)
```

   At this point, s1.xsd is invalid and cannot be used.

2. Step 2 : Register "s2.xsd" in FORCE mode:

```
dbms_xmlschema.registerSchema("s2.xsd", "<schema ..", ..., force => true)
```

   The second operation automatically compiles `s1.xsd` and makes both XML schemas valid.

CLAIMS

What is claimed is:

1. A method for managing data in a database system, the method comprising the steps of:

determining, within a database system, an appropriate database representation for storing within said database system documents that conform to an XML schema;

generating mapping data that indicates correlations between elements of said XML schema and elements of said appropriate database representation.

2. The method of Claim 1 wherein:

the step of determining an appropriate database representation includes determining, based on user-specified information, that an element of said XML schema is to be mapped to a single CLOB without generating other object types within said database system for said element; and

the step of generating mapping data includes generating data that maps said element to said single CLOB.

3. The method of Claim 2 further comprising the step of receiving said user-specified information in the form of user-specified annotations to said XML schema.

4. The method of Claim 1 wherein:

the step of determining an appropriate database representation includes determining, based on user-specified information, that a first set of subelements of an element of said XML schema is to be mapped to a single CLOB; and

the step of generating mapping data includes generating data that maps said first set of subelements to said single CLOB, and generating data that maps a second set of subelements of said element to one or more objects other than said CLOB.

5. The method of Claim 1 wherein the step of determining an appropriate database representation includes mapping datatypes associated with elements in said XML schema to datatypes supported by said database system.

6. The method of Claim 1 wherein the step of determining an appropriate database representation includes defining an SQL object type that includes attributes that correspond to elements in said XML schema.

7. The method of Claim 5 wherein the step of mapping datatypes includes the steps of:

if a particular datatype associated with an element in said XML schema is associated with a first length, then mapping said particular datatype to a first database datatype; and

if said particular datatype is associated with a second length, then mapping said particular datatype to a second database datatype, wherein the first database datatype is different than said second database datatype.

8. The method of Claim 5 wherein the step of determining an appropriate database representation includes mapping a particular element of said XML schema to a collection type supported by the database system if the particular element is defined to have a maximum number of occurrences greater than one.

9. The method of Claim 8 wherein the collection type is an array type, wherein the cardinality of the array type is selected based on the maximum number of occurrences specified for said particular database element.

10. The method of Claim 1 wherein the step of constraint determining an appropriate database representation includes defining a constraint in said appropriate database representation based upon a constraint specified in said XML schema for an element of said XML schema.

11. The method of Claim 10 wherein the step of defining a constraint includes defining a constraint from a set consisting of: a uniqueness constraint, a referential constraint, and a not null constraint.

12. The method of Claim 1 wherein:

a first datatype is associated with an element in the XML schema;

the XML schema specifies that said first datatype inherits from a second datatype; and

the step of determining an appropriate database representation includes defining within said database system a subtype of an object type, wherein said object type corresponds to said second datatype.

13. The method of Claim 1 wherein the step of determining appropriate database representation includes:

mapping a first set of elements in said XML schema to database structures that maintain each element separate in the first set separate from the other elements in the first set; and

mapping a second set of elements in said XML schema to a database structure in which all elements in said second set of elements are combined as a single undifferentiated database element.

14. The method of Claim 13 wherein the database system determines membership of said first set and membership of said second set based on directives associated with said XML schema.

15. The method of Claim 13 wherein elements in the first set of elements are selected to be in said first set based on a likelihood that said elements will be accessed more frequently than the elements selected to be in said second set of elements.

16. The method of Claim 1 wherein:

the steps of determining an appropriate database representation and generating mapping data are preformed as part of an XML schema registration operation that causes modifications within said database system; and

the method further comprises the step of automatically removing all modifications caused by said XML schema registration operation in response to encountering a particular error during said XML schema registration operation.

17. The method of Claim 1 wherein the step of determining an appropriate database representation includes determining how to break cycles in said XML schema.

18. The method of Claim 1 wherein:

said XML schema includes a cycle involving a plurality of components; and

the step of determining how to break cycles includes causing each component of the cyclic definition to holds pointers to all of its children components.

19. The method of Claim 1 wherein the step of determining how to break cycles includes causing an entire cyclic definition to be mapped for storage as a single CLOB within the database system.

20. The method of Claim 1 wherein the step of generating mapping data includes adding annotations to said XML schema, and storing said annotated XML schema within said database system.

21. The method of Claim 1 further comprising the steps of:

creating structures within a database based on said appropriate database representation; and

storing in said structures data from XML documents that conform to said XML schema.

22. The method of Claim 21 wherein the step of storing data from XML documents includes the steps of:

receiving an XML document at said database system;

identifying data, from said XML document, that is associated with individual elements of said XML schema;

storing the data associated with individual elements at locations within said structures based on

the elements associated with the data, and

the mapping data.

23. The method of Claim 1 further comprising the step of validating, within said database system, said XML schema to determine whether the XML schema conforms to an XML schema for XML schemas.

24. The method of Claim 1 wherein the step of determining is performed as part of an XML schema registration operation that is initiated in response to receiving, at said database server, said XML schema.

25. The method of Claim 1 wherein the step of determining is performed as part of an XML schema registration operation that is initiated in response to receiving, at said database server, an XML document that conforms to said XML schema.

26. The method of Claim 24 wherein:

the XML schema includes user-specified annotations that indicate how the database system should map at least one element of the XML schema; and

at least a portion of the mapping data reflects said user-specified annotations.

27. A computer-readable medium carrying instructions for managing data in a database system, the instructions comprising instructions which, when executed by one or more processors, cause the processors to perform the steps of:

determining, within a database system, an appropriate database representation for storing within said database system documents that conform to an XML schema;

generating mapping data that indicates correlations between elements of said XML schema and elements of said appropriate database representation.

28. The computer-readable medium of Claim 27 wherein the step of determining an appropriate database representation includes mapping datatypes associated with elements in said XML schema to datatypes supported by said database system.

29. The computer-readable medium of Claim 27 wherein the step of determining an appropriate database representation includes defining an SQL object type that includes attributes that correspond to elements in said XML schema.

30. The computer-readable medium of Claim 28 wherein the step of mapping datatypes includes the steps of:

if a particular datatype associated with an element in said XML schema is associated with a first length, then mapping said particular datatype to a first database datatype; and

if said particular datatype is associated with a second length, then mapping said particular datatype to a second database datatype, wherein the first database datatype is different than said second database datatype.

31. The computer-readable medium of Claim 28 wherein the step of determining an appropriate database representation includes mapping a particular element of said XML schema to a collection type supported by the database system if the particular element is defined to have a maximum number of occurrences greater than one.

32. The computer-readable medium of Claim 31 wherein the collection type is an array type, wherein the cardinality of the array type is selected based on the maximum number of occurrences specified for said particular database element.

33. The computer-readable medium of Claim 27 wherein the step of constraint determining an appropriate database representation includes defining a constraint in said appropriate database representation based upon a constraint specified in said XML schema for an element of said XML schema.

34. The computer-readable medium of Claim 33 wherein the step of defining a constraint includes defining a constraint from a set consisting of: a uniqueness constraint, a referential constraint, and a not null constraint.

35. The computer-readable medium of Claim 27 wherein:

a first datatype is associated with an element in the XML schema;

the XML schema specifies that said first datatype inherits from a second datatype; and

the step of determining an appropriate database representation includes defining within said database system a subtype of an object type, wherein said object type corresponds to said second datatype.

36. The computer-readable medium of Claim 27 wherein the step of determining appropriate database representation includes:

mapping a first set of elements in said XML schema to database structures that maintain each element separate in the first set separate from the other elements in the first set; and

mapping a second set of elements in said XML schema to a database structure in which all elements in said second set of elements are combined as a single undifferentiated database element.

37. The computer-readable medium of Claim 36 wherein the database system determines membership of said first set and membership of said second set based on directives associated with said XML schema.

38. The computer-readable medium of Claim 36 wherein elements in the first set of elements are selected to be in said first set based on a likelihood that said elements will be accessed more frequently than the elements selected to be in said second set of elements.

39. The computer-readable medium of Claim 27 wherein:

the steps of determining an appropriate database representation and generating mapping data are preformed as part of an XML schema registration operation that causes modifications within said database system; and

the computer-readable medium further comprises instructions for performing the step of automatically removing all modifications caused by said XML schema registration operation in response to encountering a particular error during said XML schema registration operation.

40. The computer-readable medium of Claim 27 wherein the step of determining an appropriate database representation includes determining how to break cycles in said XML schema.

41. The computer-readable medium of Claim 27 wherein the step of generating mapping data includes adding annotations to said XML schema, and storing said annotated XML schema within said database system.

42. The computer-readable medium of Claim 27 further comprising instructions for performing the steps of:

creating structures within a database based on said appropriate database representation; and

storing in said structures data from XML documents that conform to said XML schema.

43. The computer-readable medium of Claim 42 wherein the step of storing data from XML documents includes the steps of:

receiving an XML document at said database system;

identifying data, from said XML document, that is associated with individual elements of said XML schema;

storing the data associated with individual elements at locations within said structures based on

the elements associated with the data, and

the mapping data.

44. The computer-readable medium of Claim 27 further comprising instructions for performing the step of validating, within said database system, said XML schema to determine whether the XML schema conforms to an XML schema for XML schemas.

45. The computer-readable medium of Claim 27 wherein the step of determining is performed as part of an XML schema registration operation that is initiated in response to receiving, at said database server, said XML schema.

46. The computer-readable medium of Claim 27 wherein the step of determining is performed as part of an XML schema registration operation that is initiated in response to receiving, at said database server, an XML document that conforms to said XML schema.

47. The computer-readable medium of Claim 45 wherein:

the XML schema includes user-specified annotations that indicate how the database system should map at least one element of the XML schema; and

at least a portion of the mapping data reflects said user-specified annotations.

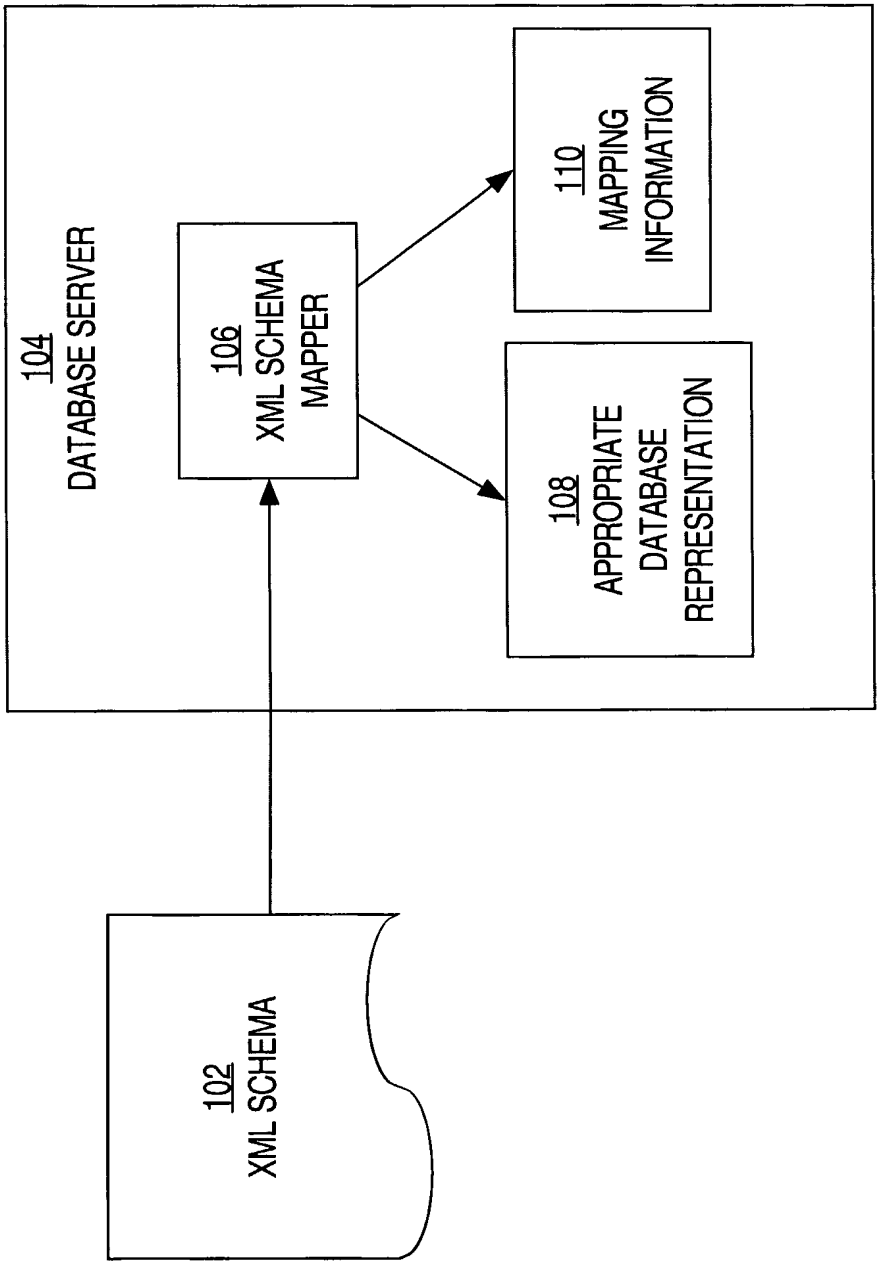

104
DATABASE SERVER
106
XML SCHEMA
MAPPER
110
MAPPING
INFORMATION
108
APPROPRIATE
DATABASE
REPRESENTATION
102
XML SCHEMA

FIG. 1

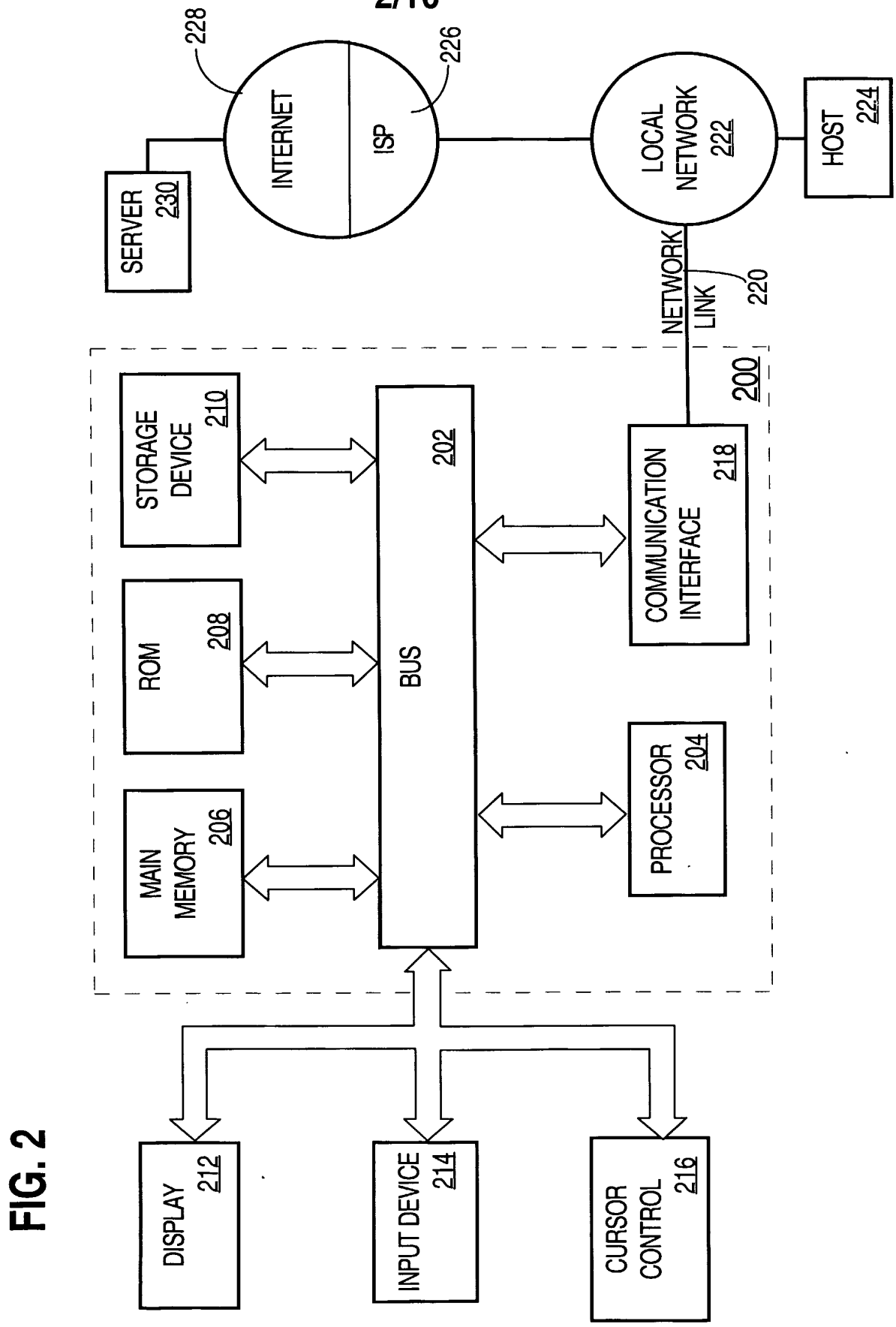

228
INTERNET
ISP
226
SERVER 230
LOCAL NETWORK 222
HOST 224
NETWORK LINK 220
200
STORAGE DEVICE 210
ROM 208
MAIN MEMORY 206
BUS 202
COMMUNICATION INTERFACE 218
PROCESSOR 204
DISPLAY 212
INPUT DEVICE 214
CURSOR CONTROL 216

FIG. 2

FIG. 3

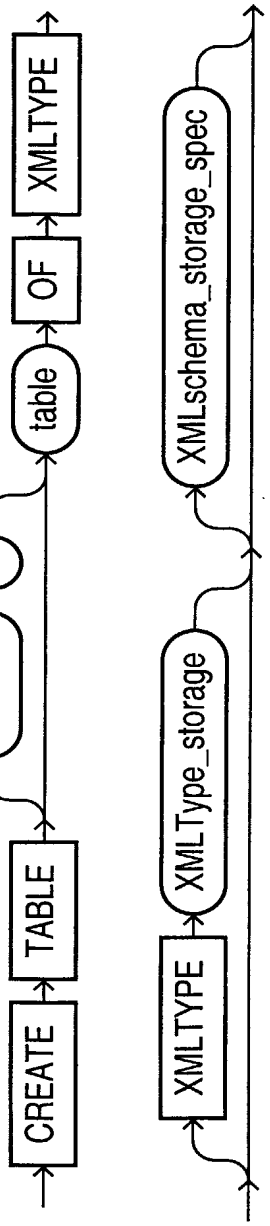

CREATE
TABLE
schema
.
table
OF
XMLTYPE
XMLTYPE
XMLType_storage
XMLschema_storage_spec

FIG. 4

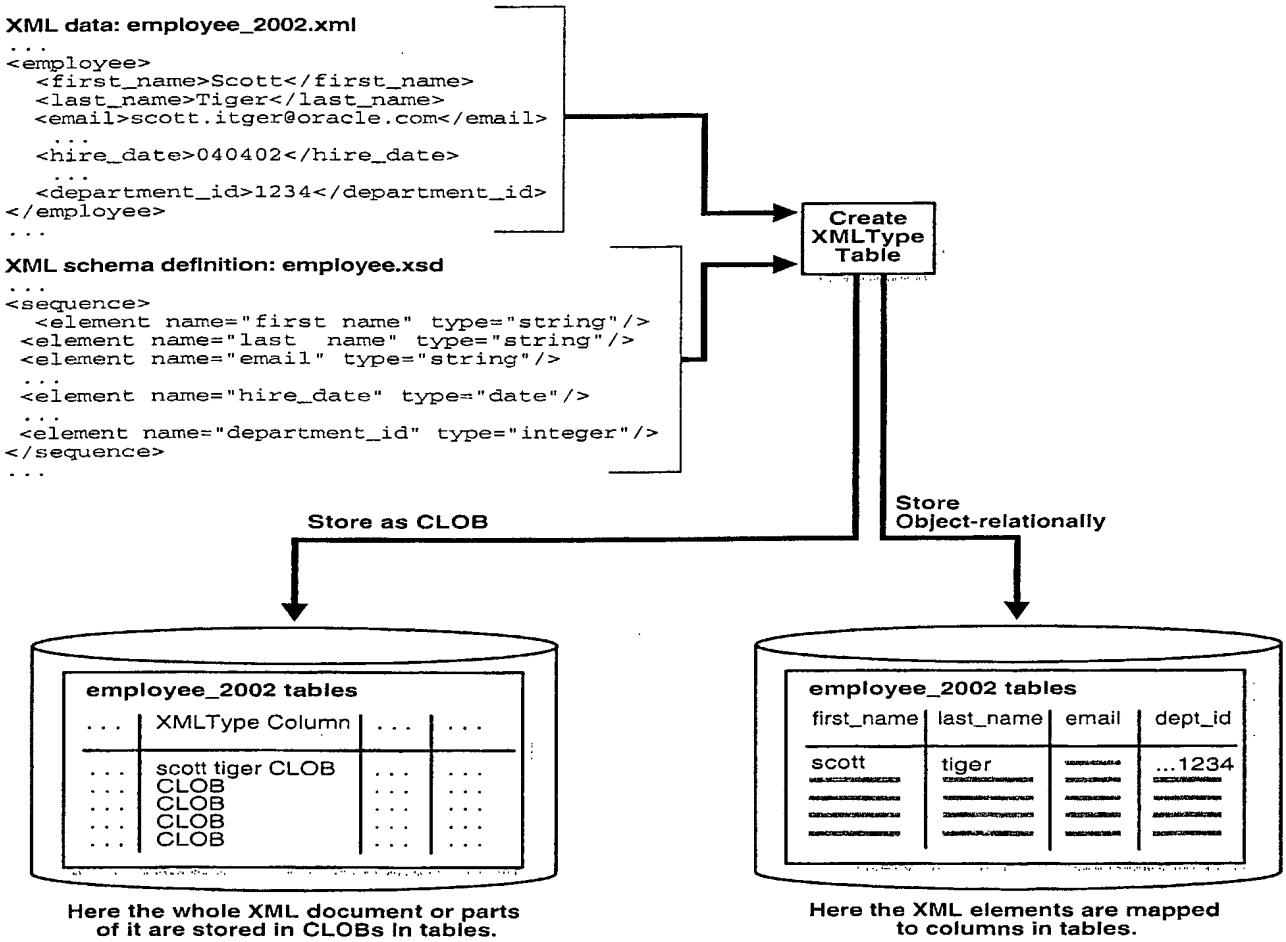

XML data: employee_2002.xml
...
<employee>
<first_name>Scott</first_name>
<last_name>Tiger</last_name>
<email>scott.itger@oracle.com</email>
...
<hire_date>040402</hire_date>
...
<department_id>1234</department_id>
</employee>
...
XML schema definition: employee.xsd
...
<sequence>
<element name="first name" type="string"/>
<element name="last name" type="string"/>
<element name="email" type="string"/>
...
<element name="hire_date" type="date"/>
...
<element name="department_id" type="integer"/>
</sequence>
...
Create XMLType Table
Store as CLOB
Store Object-relationally
employee_2002 tables
... XMLType Column ... ...
... scott tiger CLOB ... ...
... CLOB ... ...
... CLOB ... ...
... CLOB ... ...
... CLOB ... ...
Here the whole XML document or parts of it are stored in CLOBs in tables.
employee_2002 tables
first_name last_name email dept_id
scott tiger ...1234
Here the XML elements are mapped to columns in tables.

FIG. 5

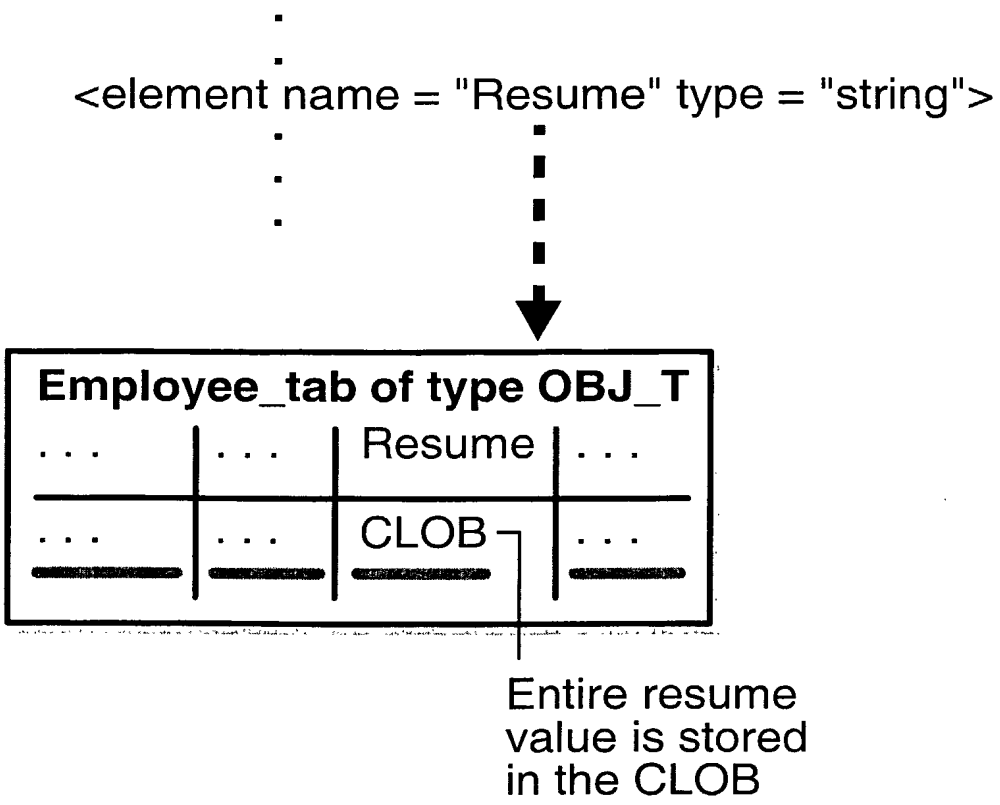

<element name = "Resume" type = "string">
Employee_tab of type OBJ_T
. . .
. . .
Resume
. . .
. . .
. . .
CLOB
. . .
Entire resume value is stored in the CLOB

FIG. 6

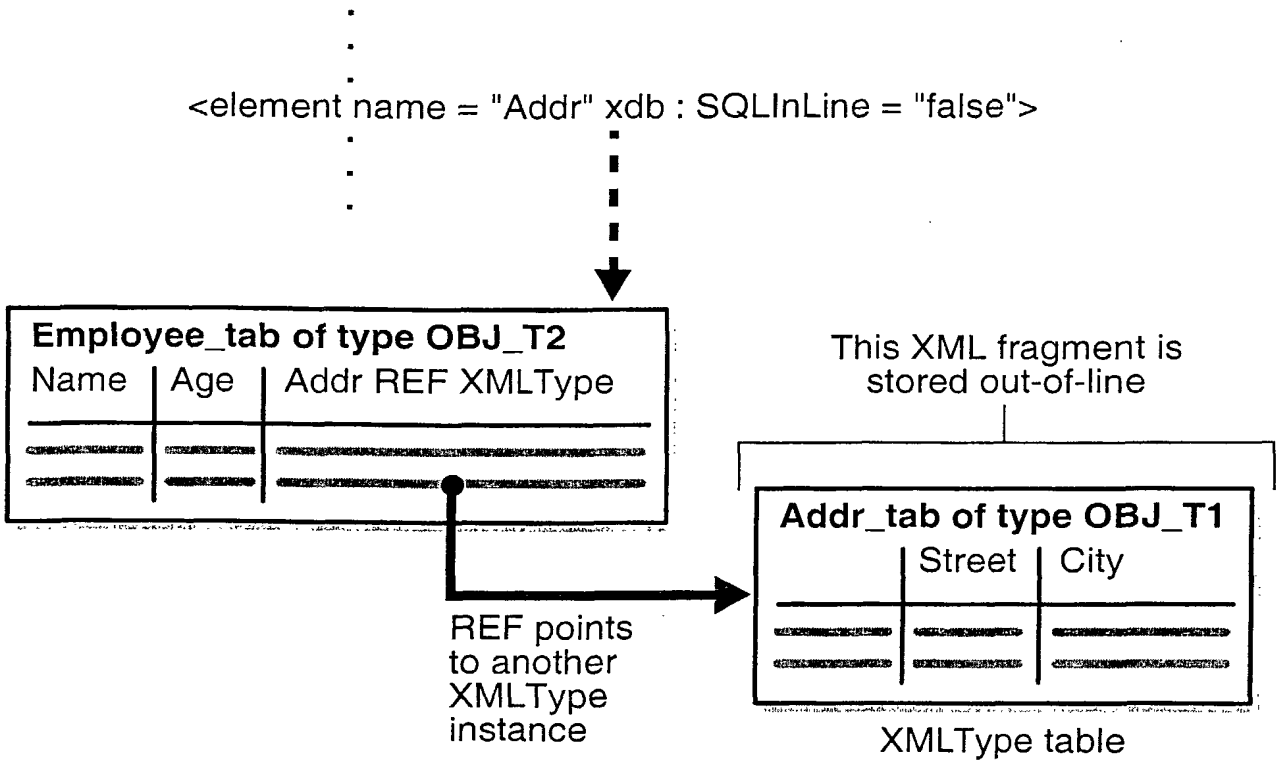

<element name = "Addr" xdb : SQLInLine = "false">
Employee_tab of type OBJ_T2
Name
Age
Addr REF XMLType
This XML fragment is stored out-of-line
Addr_tab of type OBJ_T1
Street
City
REF points to another XMLType instance
XMLType table

FIG. 7

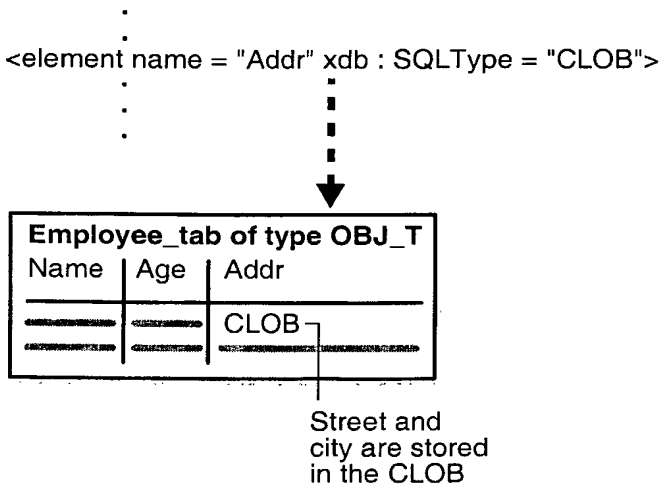

<element name = "Addr" xdb : SQLType = "CLOB">
Employee_tab of type OBJ_T
Name
Age
Addr
CLOB
Street and
city are stored
in the CLOB

FIG. 8

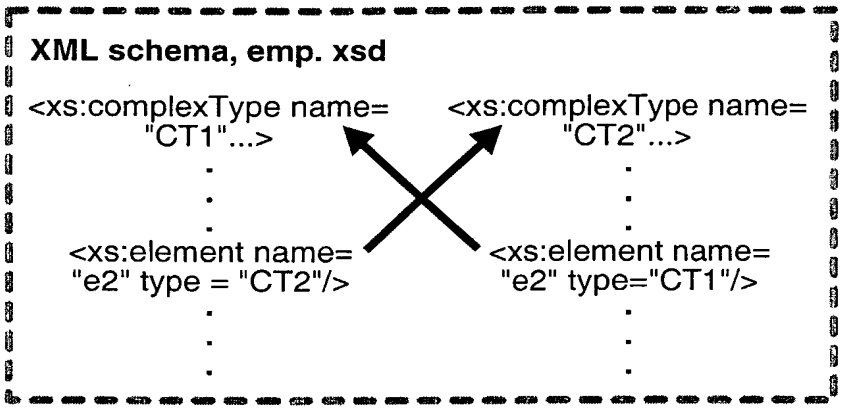

XML schema, emp. xsd
<xs:complexType name= "CT1"...>
<xs:element name= "e2" type = "CT2"/>
<xs:complexType name= "CT2"...>
<xs:element name= "e2" type="CT1"/>

FIG. 9

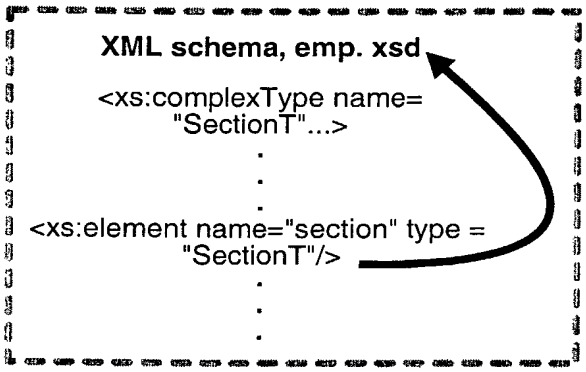
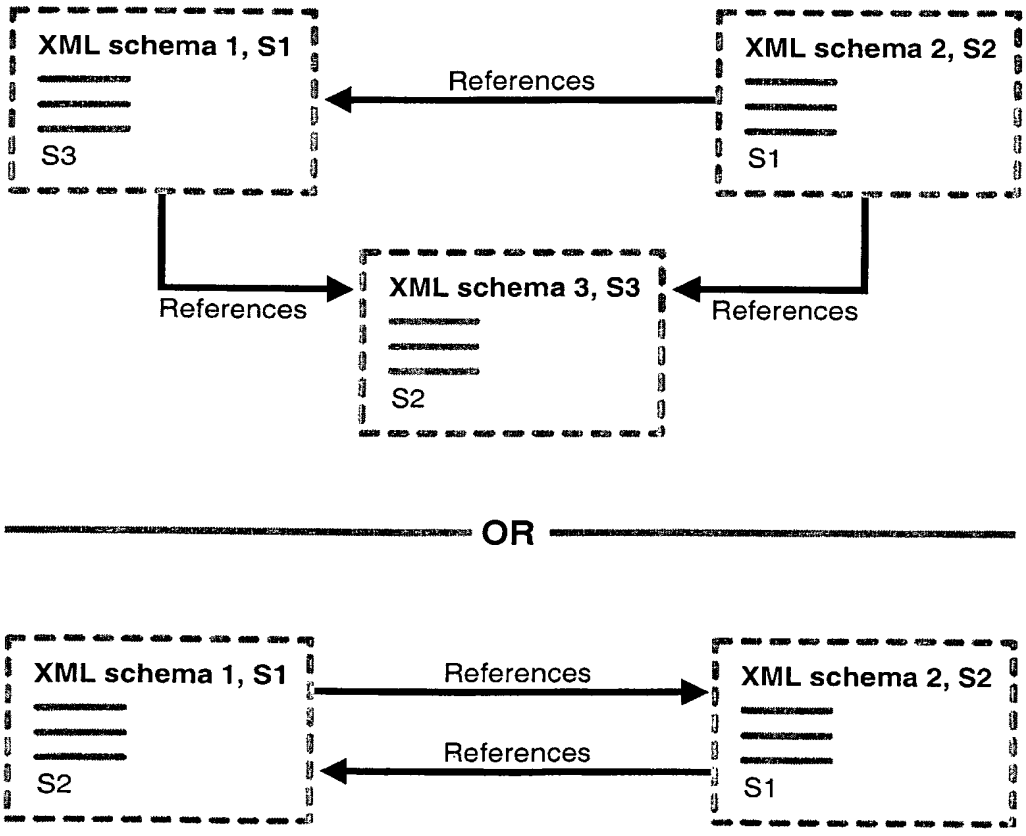

XML schema, emp. xsd
<xs:complexType name=
"SectionT"...>
<xs:element name="section" type =
"SectionT"/>

FIG. 10


XML schema 1, S1
S3
References
XML schema 2, S2
S1
References
XML schema 3, S3
S2
References
OR
XML schema 1, S1
S2
References
References
XML schema 2, S2
S1